PDM 2006

Proceedings of the
International Workshop on Parallel Data Mining

in conjunction with ECML/PKDD 2006

Giuseppe Di Fatta
Michael R. Berthold
Srinivasan Parthasarathy
(Eds.)

Berlin, Germany, 18th September 2006

Preface

Recently major processor manufacturers have announced a dramatic shift in their paradigm to increase computing power over the coming years. Instead of focusing on faster clock speeds and more powerful single core CPUs, the trend clearly goes towards multi core systems. This will also result in a paradigm shift for the development of algorithms for computationally expensive tasks, such as data mining applications. Obviously, work on parallel algorithms is not new per se but concentrated efforts in the many application domains are still missing. Multi-core systems, but also clusters of workstations and even large-scale distributed computing infrastructures provide new opportunities and pose new challenges for the design of parallel and distributed algorithms. Since data mining and machine learning systems rely on high performance computing systems, research on the corresponding algorithms must be on the forefront of parallel algorithm research in order to keep pushing data mining and machine learning applications to be more powerful and, especially for the former, interactive.

To bring together researchers and practitioners working in this exciting field, a workshop on parallel data mining was organized as part of PKDD/ECML 2006 (Berlin, Germany). The six contributions selected for the program describe various aspects of data mining and machine learning approaches featuring low to high degrees of parallelism:

The first contribution focuses the classic problem of distributed association rule mining and focuses on communication efficiency to improve the state of the art. After this a parallelization technique for speeding up decision tree construction by means of thread-level parallelism for shared memory systems is presented. The next paper discusses the design of a parallel approach for distributed memory systems of the frequent subgraphs mining problem. This approach is based on a hierarchical communication topology to solve issues related to multi-domain computational environments. The forth paper describes the combined use and the customization of software packages to facilitate a top down parallelism in the tuning of Support Vector Machines (SVM) and the next contribution presents an interesting idea concerning parallel training of Conditional Random Fields (CRFs) and motivates their use in labeling sequential data. The last contribution finally focuses on very efficient feature selection. It describes a parallel algorithm for feature selection from random subsets.

Selecting the papers included in this volume would not have been possible without the help of an international Program Committee that has provided detailed reviews for each paper. We would like to also thank Matthew Otey who helped with publicity for the workshop.

September 18, 2006

Giuseppe Di Fatta
Michael R. Berthold
Srinivasan Parthasarathy
PDM 2006 Wokshop Chairs

The International Workshop on Parallel Data Mining (PDM 2006) is held as part of the 17th European Conference on Machine Learning and the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases, September 18-22, 2006, Berlin, Germany.

Workshop Chairs

Giuseppe Di Fatta	University of Reading, UK difatta@reading.ac.uk
Michael R. Berthold	University of Konstanz, Germany berthold@uni-konstanz.de
Srinivasan Parthasarathy	Ohio State University, USA srini@cse.ohio-state.edu

Publicity Chair

Matthew Otey	Ohio State University, USA otey@cse.ohio-state.edu

Program Commitee

Gagan Agrawal	Ohio State University, USA
Pradeep Dubey	Intel Corp., USA
Mario Cannataro	University "Magna Græcia"of Catanzaro, Italy
Alok Choudhary	Northwestern University, USA
Salvatore Gaglio	University of Palermo, Italy
Robert Grossman	University of Illinois-Chicago, USA
Yike Guo	Imperial College, UK
Ruoming Jin	Kent State University, USA
Hillol Kargupta	University of Maryland, USA
George Karypis	University of Minnesota, USA
Masaru Kitsuregawa	University of Tokyo, Japan
Shonali Krishnaswamy	Monash University, Australia
Shinichi Morishita	University of Tokyo, Japan
Salvatore Orlando	University of Venice, Italy
Matthew Otey	Ohio State University, USA
Raffaele Perego	ISTI-CNR, Italy
Omer F. Rana	Cardiff University, UK
Sanjay Ranka	University of Florida, USA
Assaf Schuster	Technion, Israel Institute of Technology, Israel
Domenico Saccà	ICAR-CNR and University of Calabria, Italy
Krishnamoorthy Sivakumar	Washington State University, USA
Domenico Talia	University of Calabria, Italy
Alfonso Maurizio Urso	ICAR-CNR, Italy
Jason T. L. Wang	New Jersey Institute of Technology, USA
Ran Wolff	Technion, Israel Institute of Technology, Israel
Mohammed J. Zaki	Rensselaer Polytechnic Institute, USA
Albert Y Zomaya	University of Sydney, Australia

Table of Contents

Technical Contributions

Scalable, Adaptive Distributed Association Rule Mining Algorithms for Skewed Datasets

Marwa K. Elteir[1], Khalil M. Ahmed[2], Vijay V. Raghavan[3], Alaaeldein M. Hafez[2]

[1] Mubarak City for Scientific Research and Technology Applications, Egypt
maelteir@yahoo.com
[2]CSE, Faculty of Engineering, Alexandria University, Egypt
[3] CACS, UL at Lafayette, LA 70504, USA
raghavan@cacs.louisiana.edu

Abstract. With the existence of many distributed computing environments that have different distributed sources of huge data and several computing nodes, it is important to propose new algorithms for distributed mining of association rules. In the distributed environment, minimizing the communication cost is considered as the key challenge of discovering association rules. Although different algorithms have been proposed for such problem, they still lack some important issues, such as, scalability and data skewness. In this paper, we propose new algorithms that have less communication cost compared to the existing algorithms and also are more resilient to data skewness. Experiments generally have shown that for lightly skewed partitions, the algorithms achieve significant performance enhancement. For highly skewed partitions, the performance has been enhanced by up to 70%. In addition, they scale better with the number of nodes and it is also more resilient to data skewness, imbalanced partition sizes and message ordering.

1. Introduction

Association rule mining problem is a central problem in the field of data mining. It has a wide range of applications. Examples of theses applications include market basket analysis, health insurance and fraudulent discovery. Recently, most association rule mining algorithms assume the existence of data sources at a single location prior to the mining process. The main bottleneck that faces these algorithms is how to reduce the number of scans to cope with the huge datasets. One approach is to run a centralized algorithm on the whole dataset and incorporate techniques like sampling and partitioning [1, 2].

Another approach is to distribute datasets among several nodes and run parallel algorithms that effectively utilize the computation power, memory and disk I/O of the participating nodes [2, 3, 4, 5, 6, 7, and 8]. With the availability of several distributed computing environments that have different distributed sources of huge data and multiple compute nodes, new area named, "*Distributed Association Rule Mining (DARM)*" has emerged. The main bottleneck that faces the distributed association rule mining algorithms is the communication complexity. In the literature, a few algorithms for mining association rules in distributed environments have been

introduced. Count Distribution (CD) [3] is the first algorithm that tackles this problem. Fast Distributed Mining (FDM) [9] proposed several enhancements to the CD algorithm. However, both of them do not scale well with the number of nodes. A new algorithm named Distributed Decision Miner (DDM) [10] was recently proposed that requires less communication cost. However, a great potential still exists to benefit from the framework introduced in [10].

In [10] the distributed association rule mining problem is viewed as a decision problem in which the sites perform some kind of negotiation and at the end they are able to decide which candidate itemsets are frequent and which are not. The global support counts for frequent itemsets are collected optimally without any communication being wasted as in the case of globally infrequent but locally frequent itemsets.

For each itemset, after any site broadcasts its local support count, this count can be utilized to relax the minimum support constraint. Hence, only the sites that satisfy the relaxed minimum support constraint can broadcast its local support count. At any time if no site broadcasts its support, the itemset is considered as a infrequent itemset. For frequent itemsets, all sites should, eventually, broadcast their local support counts. Based on the above strategy, in this paper, two distributed association mining algorithms to reduce the communication costs further are proposed. We assume that the database to be studied is a transactional database. It includes a huge number of transactions each contains a set of data items. Also it is assumed that the database is horizontally partitioned and allocated to the computing nodes. Further, the intended distributed environments are assumed to be broadcasting networks.

The rest of the paper is organized as follows: In section 2, taxonomy for the distributed association rule mining algorithms is presented and the details of the most relevant algorithms are also discussed. In section 3, the proposed algorithms are introduced. In section 4, the simulation model used for the performance evaluation is described, and the results are presented and analyzed. Finally, the conclusions and some suggestions for future work are given in section 5.

2. Related work

The literature includes several communication oriented distributed association rule mining algorithms. Fig.1 presents a taxonomy for these algorithms based on the scale of the problem and the target network (whether it is broadcasting network or not). As shown in the figure most of the algorithms are designed for small scale distributed environments (number of nodes in order of tens). Only Large Scale Distributed Majority (LSD-Majority) [12], recently proposed by Schuster and Wolff, assumed a large scale distributed environment.

This study is concerned with small scale DARM with broadcasting networks. Two algorithms are targeting this area. The CD algorithm [3] is a simple algorithm. Its main advantage is that it does not exchange data tuples between the computing nodes, it only exchanges the counts. In the first scan, each node generates its local candidate itemset depending on the items present in its local partition. The algorithm obtains global counts by exchanging local counts with all other nodes. The algorithm's

communication overhead is $O(|C| \cdot n)$ at each phase, where $|C|$ and n are the size of candidate itemsets and number of data sets respectively.

Recently, Schuster and his colleagues proposed the DDM [10] which reduces communication overhead to $O(Pr_{above} \cdot |C| \cdot n)$, where Pr_{above} is the probability that a specific itemset is locally frequent in a specific partition. However the main problem of DDM is that it is not scalable with the skewness that is the normal case in the real scenarios [15]. More infrequent nodes are required to send their local support counts to convince the frequent node that certain itemset is globally infrequent. Furthermore, its performance is highly affected by the ordering of messages.

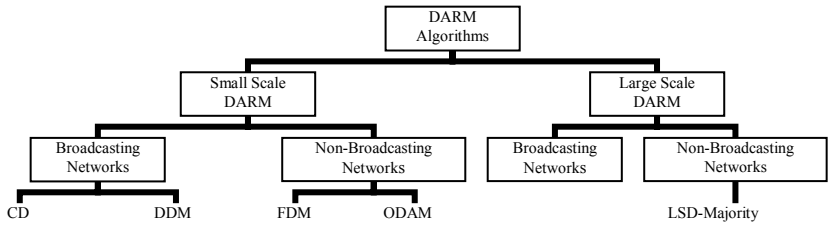


Fig. 1. A taxonomy for communication oriented distributed association rule mining algorithms

3. Proposed algorithms

3.1. Basic idea

In this paper, we propose two algorithms, the Adaptive Distributed Miner (ADM), and the Approximate Adaptive Distributed Miner (AADM). The two algorithms are based on the idea presented in this subsection. Before presenting the idea, we first introduce few notations.

Let $I= \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. An itemset X is a subset of I so that $X \subseteq I$. A transaction t is also a subset of I. A database DB is a set of transactions. Let $DB=\{DB_1,DB_2,.....,DB_n\}$ be a partition of DB into n partitions with partitions with D and D_i denoting the sizes of DB and DB_i, respectively. For any itemset X and a database partition DB_i let $S(X)_i$ be the number of transactions in DB_i that contain X. $S(X)_i$ is called the local support count of X in partition i and S(X) its global support count. For some user defined minimum support $0 \leq MinSup \leq 1$, we say that X is frequent iff $S(X) \geq MinSup * D$ and X is infrequent iff $S(X) < MinSup* D$. We also say that X is locally frequent in partition i iff $S(X)_i \geq MinSup * D_i$ and locally infrequent iff $S(X)_i < MinSup * D_i$.

For an itemset X, let the number of sites in which $S(X)_i \geq MinSup * D_i$ and broadcast their support counts be n_1. The size of the database of the n_1 sites is D_{n1} and the number of transactions that contain X in this database is $S(X)_{n1}$. The number of the remaining sites is n_2. The size of the database of the n_2 sites is $D_{n2} = D - D_{n1}$ and

the number of transactions that contain X in this database is $S(X)_{n2}$. We can state the following:

$S(X) = S(X)_{n1} + S(X)_{n2}$

Since each site i of the n_1 sites satisfy $S(X)_i \geq MinSup * D_i$, then :

$S(X)_{n1} \geq MinSup * D_{n1}$

$$\frac{S(X)_{n1}}{D_{n1}} = MinSup + \delta$$

$$\delta = \frac{S(X)_{n1}}{D_{n1}} - MinSup$$

δ is the excess in the support counts of the n_1 sites over MinSup.

Then for X to be globally frequent, at least one site i from the remaining n_2 sites should satisfy the following condition:

$$\frac{S(X)_i}{D_i} \geq MinSup - \delta * \frac{D_{n1}}{D - D_{n1}}$$

$$\frac{S(X)_i}{D_i} \geq MinSup - \left(\frac{S(X)_{n1}}{D_{n1}} - MinSup\right) * \frac{D_{n1}}{D_{n2}}$$

$$\frac{S(X)_i}{D_i} \geq \frac{1}{D_{n2}} \left(MinSup * D - S(X)_{n1}\right)$$

Clearly, $MinSup * D - S(X)_{n1}$ represents the needed support counts that should exist in the database of the n_2 sites for the itemset X to be frequent. Relaxed minimum support equals this value divided by the size, $D_{n2,}$ of the database of the n_2 sites.

3.2. Adaptive distributed miner ADM

ADM is based on Apriori algorithm and the idea described above. For each iteration k, each site i generates the candidate itemsets of size k and calculates their local support counts. For each itemset, the relaxed minimum support threshold RMinSup is initially set to the global MinSup threshold. An itemset X that has not been broadcasted and its local support count is greater than $RMinSup * D_i$ is selected from the set of these candidates. X is broadcasted with its local support count to all other sites. This process is repeated until there is no candidate that satisfies $S(X)_i \geq RMinSup * D_i$, In this case, the site has nothing to broadcast and it passes on its turn.

When site i receives a message for an itemset X, it recalculates the relaxed minimum support threshold of this itemset using the following formula:

$$RMinSup = \frac{1}{D_{n2}} \left(MinSup * D - S(X)_{n1}\right)$$ Where:

$S(X)_{n1}$ Sum of the broadcasted local support counts.

D_{n2} Size of the database of the sites that have not broadcasted their local counts.

D Size of the whole database.

It should be noted that an arriving message can cause passed sites to resume sending more messages if the local support count of any candidate becomes greater than the $RMinSup* D_i$. If a full round of passes was received from all sites, then either, in case of frequent itemsets, all of the local support counts are collected or in case of infrequent itemsets, $S(X)_i < RMinSup * D_i$.

At any pass, there may be many candidates that satisfy $S(X)_i \geq$ RMinSup * D_i and were not broadcasted. At any time, a site should select one of them to be broadcasted. The point is that if a candidate satisfies $S(X)_i \geq$ RMinSup * D_i , no arriving message can violate this satisfaction. In other words, at any time, all candidates satisfying $S(X)_i \geq$ RMinSup * D_i should be eventually broadcasted. And hence, any candidate can be selected directly and broadcasted without comparing it with the others. In case of DDM, the candidates should be ordered so that the first candidate to be sent is the one that may result in greater reduction in the overall communication cost (for more details see [10]). This let the ADM more resilient to messages ordering than DDM. The high level description of this algorithm is given in Fig.2.

For site j out of n

1- Initialize C_1={{i}: i ∈ I}, k=1, *Passed* = Φ
2- While $|C_k| > 0$
   a. Do
      i. Choose X ∈ C_k which was not yet sent and for which $S(X)_i \geq$ RMinSup * D_i, broadcast $\langle X \quad , \quad S(X)_i \rangle$ to all other sites.
      ii. If there is no such X then broadcast <*pass*>
   b. Until |Passed|=n
   c. L_k = {X ∈ C_k: the number of sites that broadcast the local support count of X is n}
   d. C_{k+1} =aprior_gen(L_k)
   e. k=k+1
3- Generate Rules($L_{1,\ldots}L_k$)

When site j receives a message M from site p:

1- If M =<*pass*> insert p into *passed.*
2- Else M= $\langle X \quad , \quad S(X)_i \rangle$
   a. If p ∈ *passed* remove p from passed
   b. Recalculate RMinSup

Fig. 2. Adaptive distributed Miner

Theorem

The communication cost of Adaptive Distributed Miner is $O(P_{large}.|C|.n)$, where n is the number of sites, |C| is the number of candidates which is the same as that considered by DDM, and P_{large} is the percentage of the sites in which the itemset is locally frequent.

Proof: The total communication cost of Adaptive Distributed Miner can be viewed as communication for itemsets that tend to be frequent and communication for itemsets that tend to be infrequent. The frequent itemsets's communication is necessary because the accurate global support counts for these itemsets are needed for the calculation of the association rules. However, the infrequent itemsets's communication is wasted.

The infrequent itemsets's communication can be viewed also as communication from sites in which the itemset is locally frequent and communication from sites in

which the itemset is locally infrequent. Based on the idea described in section 3.1, all of the sites in which the itemset is locally frequent should broadcast messages regarding their local support counts. The expected number of these messages is $O(P_{large}.|C|.n)$. To estimate the communication from the sites in which the itemset is locally infrequent, let the number of messages sent from these sites is S and the number of messages sent from the sites in which the itemset is locally frequent is L. Let their average distance from the MinSup be ε^s and ε^l respectively. Because the itemset is infrequent then the relaxed minimum support calculated after sending S and L should be greater than the support of the remaining sites (let the number of these sites be n_2) , otherwise more sites can send more messages. Also at this stage, the relaxed minimum support will be smaller than or at least equal to the initial MinSup. Assume that all partitions have the same size then:

$\frac{S(X)_{n2}}{D_{n2}} < \text{RMinSup} \leq \text{MinSup}$

Since $\text{RMinSup} \leq \text{MinSup}$ **Then**

$$\text{MinSup} - \left(\frac{S(X)_{L+S}}{D_{L+S}} - \text{MinSup}\right) * \frac{D_{L+S}}{D - D_{L+S}} \leq \text{MinSup}$$

$$\text{MinSup} - \left(\frac{L(MinSup + \varepsilon^l) + S(MinSup - \varepsilon^s)}{(L+S)} - \text{MinSup}\right) * \frac{(L+S)}{n-(L+S)} \leq \text{MinSup}$$

$$\frac{L.\varepsilon^l - S.\varepsilon^s}{n-(L+S)} \geq 0$$

$$S \leq \frac{\varepsilon^l}{\varepsilon^s} * L$$

Also since $\text{RMinSup} > \frac{S(X)_{n2}}{D_{n2}}$ **Then**

$$\text{MinSup} - \frac{L.\varepsilon^l - S.\varepsilon^s}{n-(L+S)} > \text{MinSup} - \varepsilon^{s+}$$

where $\varepsilon^{s+} > \varepsilon^s$, ε^{s+} is the average distance of the local count of n_2 sites from MinSup

$$\frac{L.\varepsilon^l - S.\varepsilon^s}{n_2} - \varepsilon^{s+} < 0$$

$$\frac{L.\varepsilon^l - (S.\varepsilon^s + n_2\varepsilon^{s+})}{n_2} < 0$$

$$L.\varepsilon^l < (S + n_2)\ \varepsilon^{s'}$$

where $\varepsilon^{s'} = \frac{S.\varepsilon^s + n_2.\varepsilon^{s+}}{(S+n_2)}$, $\varepsilon^{s'}$ is the average support deficit over $S+n_2$

$$S > \frac{\varepsilon^l}{\varepsilon^{s'}}.\ L - n_2$$

ε^l is only dependent on the distribution and not on n or C . ε^s can only increase with n and $\varepsilon^{s'}$ is dependent on both the distribution and n . Then from the inequalities shown above, for estimating S, it can be concluded that the number of messages sent from locally infrequent sites has lower or linear dependency on the number of

messages sent from frequent sites. And hence the total communication cost is $O(P_{large}.|C|.n)$.

The main factors that govern the communication wasted for infrequent itemsets are P_{large} and the ratio of infrequent vs. frequent itemsets. Clearly, as P_{large} decreases the performance of ADM becomes more efficient than DDM. P_{large} is a property of the database and can not be controlled by the algorithm. In the databases we used, P_{large} was between 0.06 and 0.25. Generally, for homogenous partitions it is expected that ADM and DDM behave similarly. As the skewness of partitions becomes larger the performance of ADM becomes more efficient.

Regarding the time complexity, DDM maintains a priority queue of candidates to be able to select the candidate to be broadcasted. Every time a message is received the corresponding candidate should be updated. Such update requires O(log|C|) time. Clearly, this time is saved in ADM. Regarding the space complexity, both of DDM and ADM need the same storage.

3.3. Approximate adaptive distributed miner AADM

The relaxed minimum support RMinSup gives an estimate of the minimum support. That approximated minimum support is realized at those sites that did not broadcast till now, to consider an itemset as a frequent itemset. To save more communication time, the negotiation about any itemset can be stopped and this itemset can be considered as a frequent itemset if RMinSup reaches to a certain level α*MinSup, where $0 \leq \alpha \leq 1$.

The main difference between the AADM algorithm and the ADM algorithm is that the convergence can occur faster if the site with the largest local support count broadcasts first. As a result the candidates list is implemented as a priority queue in which the candidates are sorted using the rating function $R(X)_i = \frac{S(X)_i}{D_i} - \text{RMinSup}$. This results in increasing the time complexity of this algorithm to O(log |C|) for each message arrival. The space complexity is the same as that of ADM.

4. Performance evaluation

4.1. Simulation model

An event-driven simulator is used to simulate the system that contains several computing nodes. Each node has its own database partition and can exchange messages with the other nodes. Fig.3 represents the simulation model of any node.

As shown in fig. 3, each node is modeled as a CPU, input queue to store the incoming messages, and output queue to store the outgoing messages. Other two data structures are included in the model to handle the mining process; the waiting queue that stores the incoming messages that are not processed in the current iteration

(messages related to iteration which is higher than the current iteration), and the candidates list that stores the candidate itemsets of the current iteration.

We choose to implement the FDDI as the broadcasting network due to the simplicity of its MAC protocol that facilitates the process of implementing it in the simulator and validating its correctness. Handling this protocol in the simulator is based on the use of a *Token Arrival Event*. Initially assumes that the event is triggered for any node in the network. If there is a frame to be transmitted, the node absorbs the token and begins the transmission. After it finishes, it triggers a *Token Arrival Event* for the next node in the network. If there are no frames to be transmitted, the node triggers the event directly for the next node and so on. This handles the core of the FDDI MAC. For the capacity allocation schema, each node sends one message containing one candidate and then blocks until the send completes. It can be assumed that there will be no asynchronous frames and the allocated capacity will be sufficient for transmitting the synchronous frames.

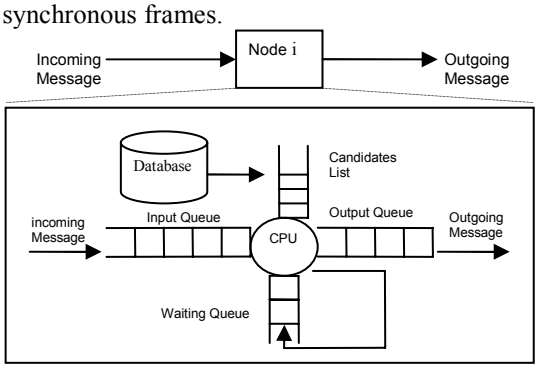


Fig. 3. Simulation model of each computing node

For the test data, the program developed in IBM Almaden research center was used for generating a synthetic transactional database. This program is available from the IBM QUEST web site (http://www.almaden.ibm.com/software/quest). It has been used in virtually every publication in mining association rules. Several parameters are used to specify the characteristics of the generated database [11]. The parameter settings used for the experimental databases are listed in table 1. The standard encoding used to name a certain database in [11] is also used here. In this encoding, the name Tx.Iy.Dzk is used to name a database for which the average transaction size is x, the average size of frequent itemset in a transaction is y, and the database size is z thousands of transactions. After the database is generated, it is partitioned among the nodes so that the skewness becomes suitable to the type of the experiment. The skewness is calculated using the entropy-based measure introduced in [14].

Table 1. Parameters settings

Database	T10I4D100k
Number of transactions	100,000
Number of items	100
Average Transaction size	10
Average size of maximal potentially frequent itemsets	4

4.2. Sensitivity to number of nodes

Figures 4a, 4b show the effect of varying the number of nodes from 4 to 32 for lightly skewed partitions (skewness 0.24) and minimum support 0.03 and 0.1. It is observed that a significant decrease in the communication cost is achieved by the proposed algorithms. Once the amount of communication is large enough, ADM performs better than DDM by 14 – 26%. And AADM performs better than ADM by 5 – 10%.

For small number of nodes, and hence small amount of communication, ADM performs better than DDM by only 6%.

It is also noticed that as the minimum support value increases from 0.03 to 0.1, the total communication cost decreases. However the performance of ADM begins to degrade only when the number of nodes becomes greater than 16. In this case the performance is 20 %.

4.3. Sensitivity to minimum support

Figures 5a, 5b show the effect of varying the minimum support values from 0.03 to 0.3 for lightly skewed partitions (skewness 0.24) and number of nodes 16 and 32. This range of support is the standard significant range for this type of databases. Above this range, the number of frequent itemsets is very small to give any meaningful comparison and below this range, the number of frequent itemsets increases rapidly.

It is noticed that, for 32 nodes system and when the amount of communication becomes large enough, ADM performs better than DDM by 20 -26 %. And for the 16 nodes system, ADM performs better than DDM by only 14% on the average.

4.4. Sensitivity to Skewness

Since most of the real distributed databases are skewed in nature, it is necessary to study the effect of varying the skewness on the performance of the proposed algorithms. This is shown in fig.6 for the 32 nodes system with Minsup = 0.1. It is noticed that a significant decrease in the communication cost is achieved by the proposed algorithms.

For skewness larger than 0.24, ADM performs better than DDM by 50 - 55%. AADM performs better than ADM by 7 - 13%. The reason is that for high skewness the number of locally infrequent itemsets increases and hence ADM has more chance for enhancing its performance over DDM. For skewness smaller than 0.24, the performance of ADM stills better than that of DDM by 20- 25% . This means that ADM is more resilient to data skewness than DDM.

4.5. Effectiveness of preemptive technique

In [10] the authors of DDM proposed another preemptive algorithm PDDM to overcome the reduced performance in case of high skewness. Another experiment is conducted to study the effect of incorporating the preemptive technique on both DDM

and ADM. Fig.7 shows the communication cost of each algorithm for different skewness values for the 32 nodes system with Minsup = 0.05.

It is noticed that for all skewness values, PDDM performs better than DDM by 7% on the average. However no considerable benefits are gained from incorporating the preemptive technique in ADM. This is expected because the order of messages (preemptive technique mainly tries to make the series of messages have a decreasing value of the rating function R) has no effects on the performance of ADM as explained in section 3.

It is also noticed that in all cases, both ADM and PADM perform better than either of the DDM and PDDM algorithms. ADM performs better than PDDM by 17-35%.

4.6. Sensitivity of AADM to relaxation factor

Fig.8a shows the effect on communication cost of varying the relaxation factor α from 0.2 to 0.01 for number of nodes 32, minimum support 0.05 and different skewness values. It is observed that AADM with α = 0.2 performs better than ADM by 5 to 14% according to the skewness values. Also it can be observed that varying the values of α from 0.2 to 0.01 does not significantly affect the performance of the algorithm.

As AADM is an approximate approach, it is necessary to study the error achieved by this approach. The error is measured as follows.

Error = $1/N * \sum_{\forall i} \frac{|S(X_i) - S'(X_i)|}{S(X_i)}$ Where

$S(X_i)$ accurate support count of itemset X_i.

$S'(X_i)$approximate support count of itemset X_i calculated by AADM.

N number of frequent itemsets.

Fig.8b shows the error achieved when α is varied from 0.2 to 0.01 for number of nodes 32, minimum support 0.05 and different skewness values. It can be observed that the error is in the range of 2 to 8 % according to the skewness. This error range is acceptable and does not significantly affect the correctness of the generated association rules.

5. Conclusions and future work

In this paper, new algorithms for association rule mining in distributed environments have been proposed. The algorithms target reducing the communication cost wasted for globally infrequent itemsets that are locally frequent at some nodes, which is considered as the *dominant* factor in the overall communication cost in many practical cases. The basic idea is based on the observation that the excess in the already broadcasted local support counts satisfying minimum support requirements, for any itemset can be utilized to relax the minimum support constraint.

An event-driven simulator is built for the performance evaluation. The performance of the new algorithms is compared to DDM algorithm [10]. The simulation results generally have shown that for lightly skewed partitions ADM achieves performance enhancement ranging from 15 to 40% in the overall communication cost. And for

highly skewed partitions, the performance enhancement reaches to 70%. Generally, the proposed algorithms are more scalable with the number of nodes and also more resilient to data skewness and message ordering than DDM. Experiments have also shown that the proposed algorithms behave well even when compared with PDDM. In addition, it was shown that AADM performs better than ADM by 5 to 14% according to the skewness and database characteristics.

More investigations have to be done to study the benefits of using other one scan algorithms such as DIC and Partition as the core of the distributed algorithm especially if the target networks are high speed networks with a reasonable bandwidth. Also, Both of DDM and ADM assume broadcast communication. When the number of computing node increases, or when the nodes are naturally divided into subgroups, it is natural to speak of hierarchic versions of these algorithms. Finally, the idea of whether the choice of the minimum support relaxation factor α can be automated or guided based on a user or domain constraint is also worth investigating.

References

1. H. Toivonen: Sampling large databases for association rules. Proceeding of the 22nd VLDB Conference, 1996.
2. A. Savasere, E. Omiecinski, and S. Navathe: An efficient algorithm for mining association rules in large databases. Proceeding of the 21st Very Large Databases Conference, Zurich , Switzerland, 1995.
3. R. Agrawal and J.C. Shafer: Parallel mining of association rules: Design, implementation and experience. Technical Report TJ10004, IBM Research Division, Almaden Research Center, 1996.
4. J. S. Park, M. S. Chen, and P. S. Yu: Efficient parallel mining for association rules. Proceeding of the 4th International Conference on Information and Knowledge Management, Baltimore, Maryland, 1995.
5. D. Cheung and Y. Xiao: Effect of Data Skewness in Parallel Mining of Association Rules. Proceeding of Pacific-Asia Conference on Knowledge Discovery and Data Mining, 1998.
6. R.C.Agarwal, C.Aggarwal, and V.V.V. Prasad: A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing Special Issue on High Performance Data Mining, 2000.
7. E. Han, G. Karypis, and V. Kumar: Scalable parallel data mining for association rules. Proceeding of ACM SIGMOD, 1997.
8. T. Shintani and M. Kitsuregawa: Hash based parallel algorithms for mining association rules. Proceeding of 4th International Conference on Parallel and Distributed Information Systems, 1996.
9. D. W. Cheung, J. Han, V. T. Ng, A. W. Fu, and Y. Fu: A Fast Distributed Algorithm for Mining Association Rules. Proceeding of 4th International Conference on Parallel and Distributed Information Systems, IEEE Computer Society Press, 1996.
10. A. Schuster and R.Wolff: Communication Efficient Distributed Mining of Association Rules. Proceedings of ACM SIGMOD, 2001.
11. R.Agrawal and R.Srikant: Fast Algorithms for Mining Association Rules. Proceeding of the 20th Very Large Databases Conference, Santiago, Chile, 1994.
12. A. Schuster and R.Wolff: Association Rule Mining in Peer-to-Peer Systems. Proceeding of the 3rd International Conference on Data Mining, 2003.

13. Schuster and R.Wolff: Communication Efficient Distributed Mining of Association Rules. Data Mining and Knowledge Discovery, 8, 171-196, 2004.
14. D. Cheung and Y. Xiao: Effect of Data Skewness and workload balance in Parallel Mining of Association Rules. IEEE, 2002.
15. R. Ayyagari, H. Kargupta: A Resampling Technique for Learning the Fourier Spectrum of Skewed Data. Proceedings of the 7th Workshop on Research Issues in Data Mining and Knowledge Discovery, ACM SIGMOD 2002. Pages 39-44

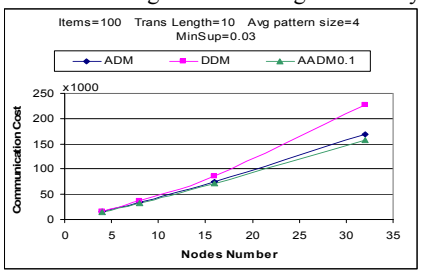


Fig. 4a. Effect of nodes number with MinSup=0.03

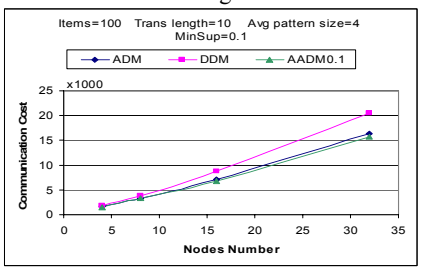


Fig. 4b. Effect of nodes number with MinSup=0.1

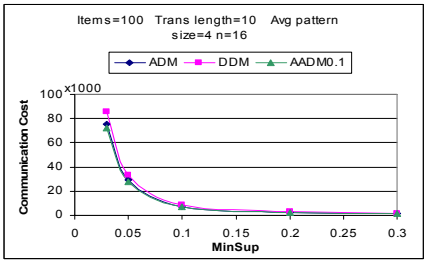


Fig.5a. Effect of minimum support n=16

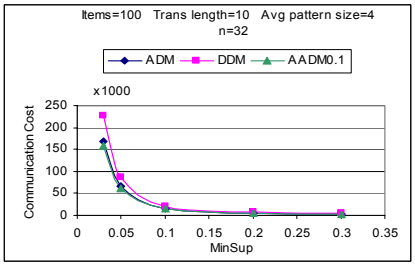


Fig.5b. Effect of minimum support n=32

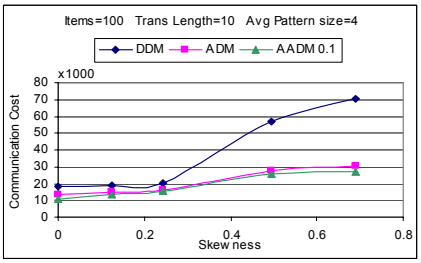


Fig. 6. Effect of skewness

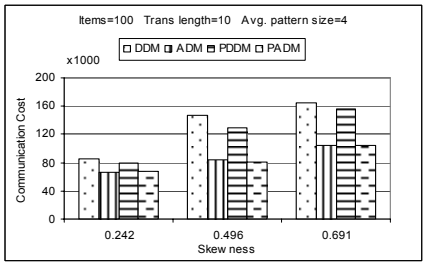


Fig. 7. Effect of preemptive technique

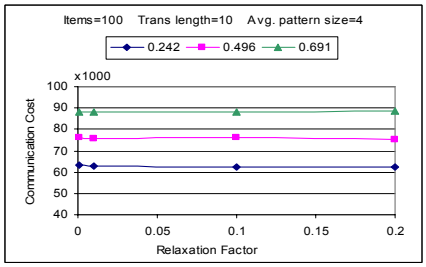


Fig. 8a. Effect of relaxation factor on communication cost

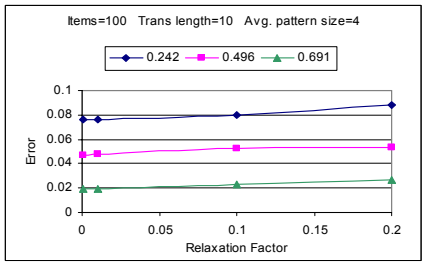


Fig. 8b. Effect of relaxation factor on error

Exploiting Thread-Level Parallelism to Build Decision Trees

Karsten Steinhaeuser, Nitesh V. Chawla, Peter M. Kogge

Department of Computer Science & Engineering
University of Notre Dame, Notre Dame, IN 46556, USA
{ksteinha, nchawla, kogge}@cse.nd.edu

Abstract. Classification is an important data mining task, and *decision trees* have emerged as a popular classifier due to their simplicity and relatively low computational complexity. However, as datasets get extremely large, the time required to build a decision tree still becomes intractable. Hence, there is an increasing need for more efficient tree-building algorithms. One approach to this problem involves using a parallel mode of computation. Prior work has successfully used processor-level parallelism to partition the data and computation. We propose to use Cray's Multi-Threaded Architecture (MTA) and extend the idea by employing thread-level parallelism to reduce the execution time of the tree building process. Decision tree building is well-suited for such low-level parallelism as it requires a large number of independent computations. In this paper, we present the analysis and parallel implementation of the ID3 algorithm, along with experimental results.

1 Introduction

Classification is one of the most important data mining tasks [1] as it arises in a large number of real-world problems. It has been studied extensively for many years [6], and a variety of classification algorithms have been presented in literature. They have evolved from simple extensions of statistical models to complex algorithms rooted in both statistics and computer science. Yet over the last two decades, *decision trees* [2][7] – one of the first and most fundamental classification techniques – have emerged as one of the most commonly used methods, and still continue to be a subject of research today.

There are several reasons why decision trees enjoy such widespread popularity. First, they are simple in concept, meaning that it is easy to understand how the clssifier is constructed from the data. Second, decision tree construction is computationally feasible (we present an analysis of the complexity in a later section). Third, a decision tree is easily interpretable by the user and can often be used directly as part of an application. Finally, decision trees have proven to perform well in a diverse range of real-world problems. Therefore, decision trees remain a very important classifier.

Despite having a relatively low computational complexity as compared to other classification algorithms, the time required to build a decision tree is still

quite high (and sometimes intractable) as datasets become extremely large. High-dimensional data with millions of examples or thousands of features is no longer uncommon, particularly for applications in the scientific domain. However, the tree building process lends itself to parallelization as there are (computationally expensive) feature-based calculations that can be performed simultaneously. Hence we want to exploit fine-grain parallelism for scaling tree learning in two dimensions (number of examples and number of features) to reduce the running time of the tree growing algorithm. Prior work exists in the parallelization of decision trees at the processor level (i.e. cluster computing), but modern architectures enable us to also take advantage of thread-level parallelism to capitalize on the inherently parallel nature of the computation.

In this paper, we present a decision tree implementation on Cray's Multi-Threaded Architecture (MTA). We analyzed the ID3 growing algorithm [7] for potential (theoretical) gains from parallelization, implemented it in C++, and performed several experiments on the MTA running in serial and parallel modes.

The remainder of the paper is organized as follows. In Section 2 we present relevant prior work in this area. In Section 3 we discuss the tree building process and analyze its complexity. Section 4 explains the unique architecture of the MTA and its technical specifications. Section 5 contains an overview of our implementation and a detailed example of parallelization. In Section 6 we present our experimental results. Section 7 outlines our future work, and in Section 8 we close with some concluding observations.

2 Related Work

SLIQ was one of the first decision tree algorithms designed specifically for high scalability [5]. It handles both numerical and nominal features, though the authors placed an emphasis on handling the former. Because the sorting of numeric attributes is one of the most expensive operations associated with classifying numeric data, SLIQ integrates a special pre-sorting procedure into the breadth-first tree growth phase. It also uses only a limited number of disk-resident data structures for metadata, while assuming that the bulk of the training data can be stored on disk. However, as datasets grow larger even these seemingly small quantities of metadata turn out to challenge memory capacities, prompting a need for memory-independent data structures and algorithms.

In [8] Shafer et al. present SPRINT, a scalable tree growing algorithm. As mentioned above, at the time memory restrictions were one of the major concerns as datasets grew larger than physical memory, which necessitated the development of efficient algorithms for disk-resident data. Therefore, SPRINT is designed to use data structures with minimal memory restrictions and as such provides fast sequential execution and exhibits good scalability (by the standards of its time). In addition, SPRINT is easily parallelizable at the processor level to further reduce execution time. However, the inter-processor communication (between the 66Mhz workstations with only 16MB of memory!) still becomes a performance bottleneck.

Murthy conducted a very thorough survey of decision tree algorithms [6]. It covers the basic method of tree construction, shortcomings, problems, solutions, variations, extensions, real-world applications and a vision for the future of decision trees.

Srivastava et al. provide a thorough theoretical treatment of the parallelization of decision tree algorithms[9]. As part of this work, the authors design and implement a hybrid approach to tree building parallelization, wherein it uses a combination of the synchronous (all processors work on the same node) and partitioned (processor subsets work on different nodes) tree construction methods. This approach is meant to balance the cost of communication overhead with the cost of load balancing at each step. Experimental results show that the algorithm scales gracefully and exhibits good speedup characteristics as the number of processors is increased.

About the same time, Zaki et al. present a parallel classification algorithm for SMP systems based on SPRINT [10][11]. The algorithm divides the computation for evaluating the best possible split between up to four processors. The algorithm shows acceptable speedup characteristics, but only for relatively small datasets. In addition, one of the great disadvantages of SMP systems is that only a single processor can access memory at any time. Because the tree building process is a very memory-intensive operation, this restriction becomes prohibitive to scalability as the processor-memory gap widens.

Later Jin and Agrawal present SPIES [4], a tree growing algorithm designed for both scalability and parallelizability. It is built on RainForest [3], a generalized framework for scaling decision tree construction. The algorithm has no memory restrictions at all and efficiently partitions the computation to minimize the amount of disk traffic required if the data is too large to fit into memory. However, this algorithm is also parallelized only at the processor level (both in a shared and distributed memory environment).

Although these parallel algorithms – SPIES in particular – exhibit reasonable performance and scalability characteristics, we argue that it is possible to further improve decision tree growing algorithms by using thread-level parallelization. The following sections will discuss the design process of such an algorithm in more detail.

3 Decision Trees

In this section, we discuss the construction of decision trees and present an analysis of the computational complexity of the tree building process.

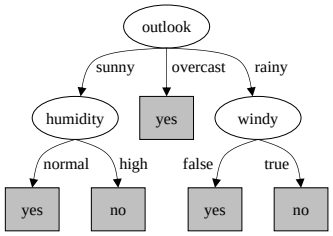
3.1 Tree Construction

Decision trees are generally constructed from *flat-file data*, or a single table in a database where each record (row) represents one example (or instance) and each column corresponds to one attribute (or feature). By convention, we let the number of examples be N and the number of features be M.

A classic example of such a dataset is shown in Figure 1(a), along with the corresponding decision tree in Figure 1(b). Here, the weather on different days is described by four attributes (outlook, temperature, humidity, windy) and the class indicates whether or not a person plays tennis on that particular day.

outlook	temperature	humidity	windy	play tennis?
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
sunny	mild	high	false	no
sunny	cool	normal	false	yes
rainy	mild	normal	false	yes
sunny	mild	normal	true	yes
overcast	mild	high	true	yes
overcast	hot	normal	false	yes
rainy	mild	high	true	no

(a) Tennis Training Data



(b) Tennis Decision Tree

Fig. 1. Example of Training Data and Decision Tree

3.2 Complexity Analysis

We shall now discuss the complexity of the ID3 tree building algorithm we chose for our implementation. First note that for this stage, we are only considering data with *nominal features* (like the tennis dataset); that is, no *continuous* or real-valued features are allowed. Including such features would have significant impact on the complexity of the algorithm, and in fact we expect even more of an improvement for this extended version. However, we chose to start with the simple case, and building a decision tree on a nominal dataset has complexity $O(MN)$ [for comparison, it is $O(M^2N)$ with continuous features].

It is also worth noting that in general the "extremely large" datasets fall into one of two categories: they either have a very large number of features and a relatively small number of examples or a very large number of examples but a manageable number of features. Our algorithm is designed to perform well in general, i.e. it should give acceptable performance for both of these extreme cases (and everything inbetween).

In order to take full advantage of the architecture, we want to parallelize in both dimensions. First, let us consider how we can parallelize along M, the number of features. Without going into too much detail of the algorithm, it should suffice to explain that at each node of the tree, all features must be evaluated and the "best" one chosen to split (by some metric, we use *information gain* with ID3). This per-feature evaluation generally involves computation on disjoint sets of data (namely only the column containing that feature) and can therefore be performed in parallel.

In addition, we can parallelize along N, the number of examples. For instance, in several places throughout the code we are required to count the occurences of

some value in a column. Using atomic increments to a counter variable, we can parallelize this operation and significantly reduce the execution time by a factor roughly equal to the number of threads available. It is precisely this low-level parallelization, which can be exploited by using multiple threads on the MTA, that is lacking from other approaches in literature.

Depending on the hardware resources available (more on this shortly) we should be able to reduce the complexity with respect to both M and N by a constant factor. If M is sufficiently small, it could even be reduced to 1, resulting a complexity of $O(N)$. Note that these reductions in no way guarantee a reduction in analytical complexity; however, the bottom line for us is execution time, and hence even a reduction by a constant factor will most definitely be of value.

4 The MTA

The Cray MTA[1] is a high-performance computer with a very unique architecture. The particular machine we are using consists of 40 processors clocked at 220Mhz. Four of these processors are mounted together on a board along with 4GB of memory per processor, resulting in a total of 160GB of main memory. This memory is *shared* among all processors, i.e. there is no notion of a per-processor local memory. In addition there is *no data caching*, which means that every load or store must actually go to memory. The processors are connected using Seastar chips, which are on-board high-performance interconnects with built-in processing and routing capability. Still, servicing a memory request takes on the order of 130 clock cycles.

One may now question how this memory latency can be overcome to achieve good performance on such an architecture. The answer lies in the design of the individual processors. More precisely, the MTA has hardware resources for *128 active threads per processor* (also called streams) and is capable of context-switching between threads in a single clock cycle. In addition, each thread can issue up to 8 concurrent memory accesses (for additional information and specs of the MTA, see http://www.cmf.nrl.navy.mil/CCS/help/mta/). Therefore, we can mask the memory overhead by using enough threads to keep a processor occupied while memory references are being serviced. Our goal then is to have not only a few or even a few dozen, but ideally on the order of hundreds or thousands of threads executing at any time to ensure that the processor remains saturated, keeping it busy with "real work" while memory accesses are serviced. And because the memory accesses required by the decision tree construction process have very poor locality, this model of multiple simultaneous accesses enjoys a great advantage over a traditional cache-based system, which incurs frequent cache miss penalties.

[1] http://www.cray.com/products/programs/mta_2/

5 Implementation

In this section we briefly discuss the ID3 algorithm, our implementation, and a detailed example of a parallelized operation.

5.1 The ID3 Decision Tree Algorithm

We chose to use ID3 because it is very fundamental in the sense that it only performs the most basic operations of decision tree construction: creating a node, testing the termination conditions, selecting the best split, dividing the data, and calling itself recursively. Figure 2 shows the pseudocode of the ID3 decision tree algorithm.

```
ID3(Examples, Attributes, TargetAttribute)

Create RootNode for the tree
if all members of Examples are in the same class C
        then RootNode = single-node tree with label = C
else if Attributes is empty
        then RootNode = single-node tree with label = most common value of Target_attribute in
                                                      Examples;
else
        A = element in Attributes that maximizes InformationGain(Examples, A)
A is decision attribute for RootNode
for each possible value v of A
        add a Branch below RootNode, testing for A = v
        Examples_v = subset of Examples with A = v
        if Examples_v is empty
                then below Branch add Leaf with label = most common value
                                                        of Target_attribute in Examples;
        else
                below Branch add Subtree ID3(Examples_v, Attributes - {A}, TargetAttribute);

return RootNode;
```

Fig. 2. Pseudocode of ID3 Decision Tree Algorithm

5.2 An Example of Parallelism

Looking at the pseudocode (Figure 2), one should notice a call to an external procedure *InformationGain*. The purpose of this function is to determine the best feature to split on. While such a simple function call looks rather harmless, it is in fact the most computationally expensive step of the algorithm. Therefore, we present an in-depth look at this operation and explain how parallelism helps us reduce its execution time.

Computing the information gain for a feature involves computing the entropy of the target attribute for the entire dataset and subtracting the conditional

entropies for each possible value of that feature. Therefore, the entropy of a subset is a fundamental calculation to compute information gain, making it the focus of our discussion. The entropy calculation requires a frequency count of the target attribute by feature value, so the straightforward way of computing the entropy for all possible values of a feature works as follows: select all examples with some feature value v, then count the number of occurences of each class within those examples, and compute the entropy for v. This step is repeated for each possible value v of the feature, which can easily be done using a for-loop.

The entropy of a subset can actually be computed more easily by constructing a count matrix, which tallies the class membership of the training examples by feature value. An example of such a count matrix for the feature *outlook* is shown in Figure 3.

	yes	no
sunny	2	3
overcast	4	0
rainy	3	2

Fig. 3. Example of Count Matrix for Outlook

Note that it only takes a single pass over the data to construct this count matrix for any feature (and in fact the count matrices for *all* features!). Once the matrices are constructed, we can easily use them as look-up tables to compute all the entropies and the information gain of each feature.

Now let us bring parallelism back into the picture and show how the information gain calculation benefits from it. Recall that we want to parallelize the computation along both the number of features and the number of attributes. To this end, we created a three-dimensional array to store the count matrices for all features in a single data structure. Instead of looking at each field in the data array sequentially, we can now use a separate thread to inspect each element and increment the corresponding variable in the appropriate count matrix.

We can actually perform one additional optimization step on this process. Notice that with many threads attempting to increment a limited number of counters, atomic add operations must be used. The necessary synchronization can be quite expensive. Fortunately, the MTA provides an instrution called INT_FETCH_ADD, which not only ensures that the increment is atomic but even performs the increment with a single instruction. Thus, we have taken the information gain calculation from a complex operation involving the copying of data and additional function calls – which is often implemented as described earlier – to a relatively simple tallying process that can be highly optimized.

To further illustrate the parallelization and optimization process, we have included the pseudocode for the information gain calculations. Figure 4(a) shows the operations for the straightforward serial version. Notice that at each iteration of the first nested loop, the entire training data must be scanned to construct the subset of data corresponding to the current feature value. In some imple-

```
for each attribute A
        Gain[A] = Entropy(Examples)
        for each possible value v of A
                Examples_v = subset of Examples with A = v
                for each example in Examples_v
                        CountMatrix[TargetAttribute]++
                Gain[A] = Gain[A] – Entropy(CountMatrix)
select A such that Gain[A] is maximized
```

(a) Serial

```
for each example E
        for each attribute A
                INT_FETCH_ADD(CountMatrix[A][A.value][TargetAttribute])
for each attribute A
        Gain[A] = Entropy(Examples)
        for each possible value v of A
                Gain[A] = Gain – Entropy(CountMatrix[A][v]);
select A such that Gain[A] is maximized
```

(b) Parallel

Fig. 4. Pseudocode for Information Gain Calculation

mentations this subset is even copied to a separate data structure for processing, which is an expensive operation.

In contrast, Figure 4(b) shows the operations for the parallelizable version. It uses two separate loops, each of which can be performed in parallel. In the first loop, a separate thread can be used for each combination of attribute A and value v, effectively making the construction of the count matrix a constant-time operation. Of course the number of threads is limited and there is some memory latency, but with hundreds of simultaneous lookups those penalties are easily overcome. Similarly in the second loop, one thread of computation can be used per attribute, each spawning additional threads for each possible value used by the gain calculations in the nested loop. Given that the number of attributes can be on the order of thousands and the number of examples in the hundreds of thousands of more, using this thread model will achieve the desired goal of issuing enough memory references to make up for the latency and outperform the serial version.

6 Experiments and Results

This section introduces the datasets, explains the experimental setup, presents the results, and discusses them in the context of the analysis provided earlier.

6.1 The Data

We selected two datasets for our experiments; the Forest CoverType dataset from the UCI KDD Archive[2], which contains 581,000 examples of 44 binary fea-

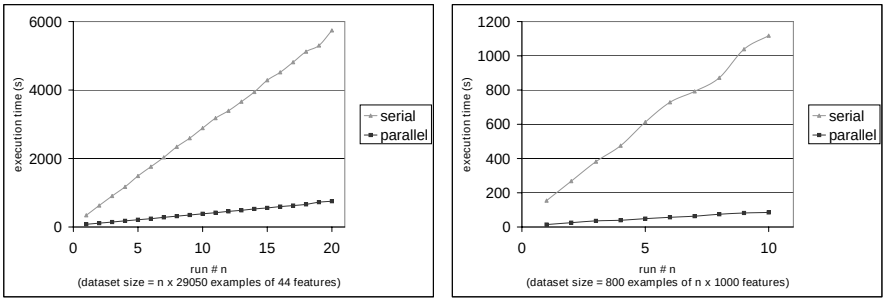
[2] http://kdd.ics.uci.edu/

tures and 7 classes (after removing the continuous features); and a gene dataset called Dorothea[3], which contains only 800 examples, but has 10,000 features and 2 classes. These datasets represent the two extreme situations mentioned earlier, wherein there are either many examples and a relatively small number of features, or few examples with a large number of features.

To test the scalability characteristics along the number of examples N, we constructed 20 datasets of increasing size from the CoverType data. Similarly, to test the scalability characteristics along the number of features M, we used the Dorothea dataset to construct 10 datasets with an increasing number of features (using the first 1,000 features, then 2,000, etc).

6.2 Experimental Setup

We ran our decision tree code the MTA in serial mode (i.e. forcing it to use a single thread of execution, making it very similar to a regular processor) and in parallel mode (allowing up to 128 active threads of execution) for both the CoverType and the Dorothea dataset. The results are shown in Figures 5(a) and 5(b), respectively. Notice that the size of the dataset (plotted on the x-axis) increases with each run for both datasets, but in a different dimension.



(a) CoverType (b) Dorothea

Fig. 5. Execution Time vs. Training Set Size

6.3 Discussion of Results

In this section, we discuss two important observations in the result data. Let us begin with the more obvious of the two.

Execution Time For both of the CoverType and the Dorothea data, the execution time is significantly lower in parallel execution than in serial execution.

[3] http://clopinet.com/isabelle/Projects/NIPS2003/

This confirms our main hypothesis, namely that we can exploit thread-level parallelism to reduce the execution time of the decision tree building process.

Note that for the CoverType data, the MTA reported using only an average of 11.8 threads for the entire tree building process and 20.8 threads for the Dorothea data. These values are a large part of the explanation why the execution times are only reduced by a factor of about 5 in the case of CoverType, and a factor of 10 for Dorothea. One would think that the execution time is cut by a factor roughly equal to the total number of threads available. However, this is actually not the case because (i) not all of the code can be parallelized, so there will be sections that run entirely sequentially and (ii) even in parallel sections there are not necessarily enough instructions to saturate all threads, especially as computation proceeds further down the tree and the dataset at each node becomes smaller.

Nonetheless, we get a significant performance improvement when using the parallel mode of execution, as illustrated by the raw time savings: we can process the full CoverType dataset in under 13 minutes in parallel mode versus 96 minutes in serial mode; for the Dorothea dataset the contrast is equally noticable with 85 seconds versus 19 minutes.

Scalability The more subtle observation in the results is that the execution time scales more gracefully in parallel mode of execution as compared to the serial mode. It is difficult to see in the graphs, so we have reproduced a condensed version of the data in Figure 6, which shows the ratio of the execution times between the serial and parallel modes for the same dataset sizes. Here we see that as the size of the dataset increases, the execution time for the parallel mode is not only absolutely faster, but it also scales more gracefully as it does for the serial mode.

Forest CoverType	
Fraction of Data	*Ratio of Execution Time Serial / Parallel*
5%	4.30
50%	7.48
100%	7.63

Dorothea	
Fraction of Data	*Ratio of Execution Time Serial / Parallel*
10%	10.78
50%	12.55
100%	13.16

Fig. 6. Serial vs. Parallel Scalability Characteristics

7 Future Work

This work merely lays the foundation for a more extensive study of using thread-level parallelism in conjunction with decision trees. There are still several areas which we would like to explore.

First, it should be possible to further optimize the code by modifying it to allow for more fine-grain parallelsim. In addition, we could introduce an entirely

new level of parallelism by making the recursive call of the main function a parallel operation, thereby enabling the MTA to evaluate the data and continue building the tree at multiple nodes simultaneously.

Second, we would like to work with larger real-world datasets. The largest dataset among those used in this study only produced a program footprint of 1.5 GB, which can still fit into main memory of a well-equipped workstation. However, we are garnering datasets that reach well into the gigabytes – such as the protein database data – and at this point the MTA's 160GB of shared memory should have a tremendous advantage. Our goal is to run a large suite of experiments on datasets that push both dimensions to the extreme (examples and features).

Finally, we are looking to extend the functionality of the code to include continuous values, which poses new challenges but also enables the use of parallelism to reduce the complexity. This would make the code compatible with other popular decision tree implementations and allow direct performance and scalability comparisons.

8 Conclusion

In this paper, we have theoretically established and empirically confirmed the hypothesis that exploiting thread-level parallelism can significantly reduce the time of the decision tree building process. By using example datasets from the extreme ends of the spectrum, we showed that our parallel decision tree implementation is consistently faster than the sequential version and scales more gracefully as the size of the input increases (both in number of examples and number of features).

Having successfully produced these important initial results, we are continuing this line of research to produce highly efficient and extremely scalable decision tree building code by taking full advantage of the inherent parallelism in the tree building process and the unique architecture and features of the Cray MTA.

Acknowledgements This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. BNCH3039003. We would sincerely like to thank Cray Inc. for their permission to use the MTA at the SDSC, especially John Feo for his tireless assistance and and expertise in parallel programming on the MTA.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Database Mining: A Performance Perspective. IEEE Transactions on Knowledge and Data Engineering **5** (1993).
2. Breiman, L., Friedman, J. H., Olshen, R. A., Stone, C. J.: Classification and Regression Trees. Wadsworth Adv. Book Prog., Belmont, CA (1984).
3. Gehrke, J., Ramakrishnan, V., Ganti, V.: RainForest – A Framework for Fast Decision Tree Construction of Large Datasets. Journal of Data Mining and Knowledge Discovery **4** (2000) 127–162.

4. Jin, R., Agrawal, G.: Communication and Memory Efficient Parallel Decision Tree Construction. 3^{rd} SIAM Int'l Conference on Data Mining, San Francisco, CA (2003).
5. Mehta, M., Agrawal, R., Rissanen, J.: SLIQ: A Fast Scalable Classifier for Data Mining. 5^{th} Int'l Conference on Extending Database Technology, Avignon, France (1996).
6. Murthy, S. K.: Automatic Construction of Decision Trees from Data: A Multi-Disciplinary Survey. Journal of Data Mining and Knowledge Discovery **2** (1998) 345–389.
7. Quinlan, J. R.: Induction of Decision Trees. Journal of Machine Learning **1** (1986) 81–106.
8. Shafer, J., Agrawal, R., Mehta, M.: SPRINT: A Scalable Parallel Classifier for Data Mining. 22^{nd} Int'l Conference of Very Large DAtabases, Bombay, India (1996).
9. Srivastava, A., Han, E.-H., Kumar, V., Singh, V.: Parallel Formulations of Decision-Tree Classification Algorithms. Int'l Conference on Parallel Processing, Minneapolis, MN (1998).
10. Zaki, M., Ho, C.-T., Agrawal, R.: Parallel Classification for Data Mining on a SMP Systems IBM Technical Report, Almaden Research Center, San Jose, CA (1998).
11. Zaki, M., Ho, C.-T., Agrawal, R.: Parallel Classification for Data Mining on a Shared-Memory Multiprocessors. 15^{th} Int'l Conference on Data Engineering, Sydney, Australia (1999).

A Hierarchical Distributed Approach for Mining Molecular Fragments

Christoph Sieb[1], Giuseppe Di Fatta[2], and Michael R. Berthold[1]

[1] ALTANA Chair for Bioinformatics and Information Mining
Department of Computer and Information Science, University of Konstanz
Box M 712, 78457 Konstanz, Germany
{sieb,berthold}@inf.uni-konstanz.de
[2] School of Systems Engineering, University of Reading
Reading RG6 6AY, United Kingdom
difatta@reading.ac.uk

Abstract. Recently, two approaches have been introduced that distribute the molecular fragment mining problem. The first approach applies a master/worker topology, the second approach, a completely distributed peer-to-peer system, solves the scalability problem due to the bottleneck at the master node. However, in many real world scenarios the participating computing nodes cannot communicate directly due to administrative policies such as security restrictions. Thus, potential computing power is not accessible to accelerate the mining run. To solve this shortcoming, this work introduces a hierarchical topology of computing resources, which distributes the management over several levels and adapts to the natural structure of those multi-domain architectures. The most important aspect is the load balancing scheme, which has been designed and optimized for the hierarchical structure. The approach allows dynamic aggregation of heterogenous computing resources and is applied to wide area network scenarios.

1 Introduction

Due to the enormous amount of data created by many of today's transactional applications, it has become necessary to parallelize the corresponding mining algorithms to attain reasonable response times. One of these applications in the field of drug discovery is related to the *High Throughput Screening* (HTS) technology. The HTS process is widely used to identify potential compound candidates for further research in the drug discovery process. HTS is able to screen more than 100,000 compounds a day for several activities, such as inhibition of HIV or cancer cells. The screened compounds are recorded in a transaction-like database together with their activity level. As the number of candidate compounds is extremely large, it is useful to extract features from the active compounds and use them to reduce the number of relevant candidates.

Key features are those molecular fragments that occur frequently in active compounds but infrequently in non-active ones, and therefore represent promising starting points. These discriminative fragments can be extracted by modeling

the compounds as undirected labeled graphs and applying *Frequent Subgraph Mining* (FSM) algorithms [9] to them. Even though the available FSM algorithms use sophisticated methods to speed up the mining process, the scalability issue can only be solved by increasing the computational power of the underlying system. Increasing the power of a single processor machine is limited by current technology and physical laws. One possible approach to overcome these limitations is to partition the original problem into smaller subtasks and allocate them to several processors.

Large companies and institutions typically deploy many ordinary, heterogeneous desktop PCs and servers at different locations with several security policies. They are often underutilized and therefore offer a large-scale computing resource pool. The challenge is to make this pool accessible for computing-intensive applications to solve problems without the need of expensive special hardware.

The first work on distributed mining of frequent molecular fragments [5] describes a centralized master/worker approach, which partitions the induced depth-first search tree to distribute the mining task. Very often such search problems exhibit a highly irregular tree shaped computation, and, in some cases as in molecular fragment mining, the complexity of subtasks cannot be estimated. In this case it is essential to adopt a dynamic load balancing policy. Typically, for FSM algorithms, the search space representation in memory is much bigger than the database size and thus, parallel approaches distribute the induced search tree instead of the database. The second approach, designed as a completely distributed peer-to-peer system [6], solved the inherently scalability problem due to the bottleneck of the central master. Both approaches require direct channel communication to exchange messages. However, in many real world applications the potential computing nodes are not always directly accessible from each other due to security restrictions. In those environments both approaches cannot exploit the potential computing power and therefore represent an *architectural bottleneck*.

In this work we present the first hierarchical distributed system for FSM and other highly skewed search tree problems applied in the context of ordinary computer technology. The hierarchical system aligns to the inherent hierarchy of those multi-domain networks to overcome the described drawback of a centralized or peer-to-peer system. The load balancing scheme takes into account the specific challenges of highly skewed search tree problems and the hierarchical structure respectively. Furthermore, it maintains locality to keep the system scalable. The tests show that the system performs similar to the centralized approach but additionally enables access to multi-domain clusters restricted by real world security policies.

The rest of this paper is structured as follows. The next section discusses related approaches to hierarchical distributed systems and FSM. In section 3, we briefly describe a concrete sequential FSM algorithm on which our distributed approach is tested. In section 4, we introduce the centralized master/worker approach, followed by section 5, which presents the architecture of the hierarchical

system and describes the hierarchical load balancing scheme. Section 6 describes the experiments we conducted to verify the performance of the hierarchical approach. Finally, we provide concluding remarks.

2 Related work

Many dynamic load balancing (DLB) algorithms for irregular problems have been proposed in literature and their properties have been studied [7]. In the field of hierarchical distributed computing, Antonis et al. describe in [2] a hierarchical load balancing scheme that needs to know the number of participating nodes in advance and builds up a logical binary tree.

In [4] Dandamudi and Lo present a load sharing policy for identical nodes by arranging them in a hierarchical structure. This work addresses hierarchical distributed systems and, in particular, sender and receiver initiated load balancing techniques. In [8] a hierarchical load balancing scheme is described that works close to the operating system. The scheme is implemented on a massive parallel system where the processors are connected by high speed networks. The described load balancing scheme, however, needs to know the current load level, which is not known in search tree problems with highly skewed data.

Finally in [10], a whole framework is proposed to provide a common platform for distributed applications. In this framework, it is necessary to know the problem size of a task to enable good load distribution. Unfortunately, this is impossible for most data mining problems as the size of the underlying search space is unknown a priori.

3 Molecular fragment mining

The problem of selecting discriminative molecular fragments in a set of molecules can be formulated in terms of frequent subgraph mining in a set of graphs. Molecules are represented by attributed graphs, in which each vertex represents an atom and each edge a bond between atoms. Each vertex carries attributes that indicate the atom type and a possible charge and each edge carries an attribute that indicates the bond type. Frequent molecular fragments are subgraphs that have a certain minimum support in a given set of graphs, i.e., are part of at least a certain percentage of the molecules. Discriminative molecular fragments are contrast substructures that are frequent in a predefined subset of molecules and infrequent in the complement of this subset. In this case, two parameters are required: a minimum support ($minSupp$) for the focus subset and a maximum support ($maxSupp$) for the complement.

The algorithm organizes the space of all possible fragments in an efficient search tree. An example of such a search tree is depicted in Figure 1. The algorithm is based on an exhaustive depth-first search strategy. Each node of the search tree represents a candidate frequent fragment. A search tree node evaluation comprises the generation of all the embeddings of the fragment in the molecules. An embedding of a fragment consists of references into a molecule

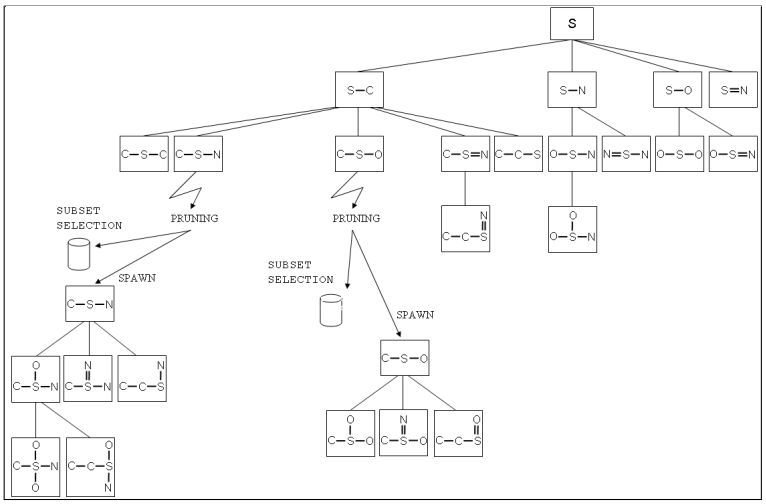


Fig. 1. Search tree partitioning

that point out the atoms and bonds that form the substructure. The embedding list allows both a fast computation of the fragment support in the active and inactive molecules and a fast extension to bigger fragments. When a fragment meets the minimum support criterion, it is extended by one bond to generate new search tree nodes. When the fragment meets both criteria of minimum support in active molecules and maximum support in the inactive molecules, it is then reported as a discriminative frequent fragment.

The search starts from a root node with a single atom and is iterated for each frequent atom type. The algorithm prunes the DFS tree according to three criteria. The support-based pruning exploits the anti-monotone property of fragment support. The size-based pruning exploits the anti-monotone property of fragment size. And, finally, a partial structural pruning is based on a local order of atoms and bonds. For further details of the algorithm we refer to [3].

After introducing the underlying application algorithm, the next section describes the centralized master/worker approach to introduce the basic DLB scheme.

4 The master/workers approach

There are two aspects in parallel and distributed systems that influence scalability, overall performance and flexibility. The first aspect is the logical communication topology, while the second aspect concerns the load balancing scheme. The logical communication topology is important with respect to scalability issues and the flexibility to adapt to administrative policies, e.g. secure subnets.

Nevertheless, load balancing must be tightly incorporated and must respect the logical structure.

In the master/workers approach, the topology is a star scheme, where the worker nodes perform the actual mining task and the master is responsible for distributing the workload equally. The distribution of the mining task is done by partitioning the search space, i.e. each worker explores a different part of the search space and the master node merges partial results. Search space partitioning is performed by pruning a branch of the depth-first search tree induced by the mining algorithm. To this aim, an external description of the pruned search node has to be generated and donated to an idle worker (see Figure 1). This description must be appropriate to set up the same state of the mining process for the receiving worker to explore the search subtree. A donated search node describes a molecular fragment that occurs in a subset of the molecular database. A pruned fragment with the information necessary to restart the computation in any of the participating nodes is subsequently referred to as a job. Each computing node has its own replica of the dataset. In order to decrease the computation overhead of re-embedding the fragment in the molecules, the ID list of the supporting molecules can be included in the description (subset selection).

The adopted DLB scheme is a receiver-initiated policy because data mining applications are typically computation bounded problems. The master node manages a pool of unprocessed jobs from which it serves child requests.

The master node sends job pruning requests to worker nodes (called donators) once the job pool size J is below a given threshold. This threshold is derived from the number of worker nodes C and a relative threshold value α_L called the relative job pool threshold. Thus, a soft state request is periodically sent while the following boolean expression holds $J < J_L$, where $J_L = \alpha_L \cdot C$ is the absolute job pool threshold. If the number of jobs is below the threshold J_L the master node sends j job requests to its children, whereas j is calculated by the difference of the job pool size J and the absolute pool threshold J_L plus the number of pending requests c: $j = J_L - J + c$

The master node chooses a worker with a previously assigned job that can donate a part by pruning its search tree. In dynamic load balancing schemes the idea is to ask the node with most work. Unfortunately, in this kind of search problems the actual complexity of a subtask is not known in advance. The load can only be estimated by the heuristic that older assignments represent bigger jobs. To decrease the probability of a bad selection and to avoid too many requests to a single node the requester selects the donator by a Ranked Random Polling [6].

When a worker has received a job request, it applies two heuristics which increase the probability of pruning a big job, referred to as *pruning heuristics*. The first one applies a focus support threshold $suppTH$, which is greater than the minimum focus support $minSupp$. Fragments with larger focus support $supp_F$ have a higher probability of being further extended instead of being

discarded due to the downward closure property. Therefore, only fragments for which $supp_F \geq suppTH$ holds are considered for pruning.

The second heuristic exploits the *local order* defined in the sequential algorithm [3] to identify larger jobs. This order defines which atoms of the current fragment can be extended. A fragment with many extendible atoms creates a larger sub tree in the search space. Thus, fragments can only be pruned if the number of extendible atoms over the total number of atoms is larger than a threshold t_L, $0 \leq t_L \leq 1$.

Whenever a job is finished at a worker node, the result is reported to the master node, which merges the partial results. Once the master's job pool is empty and no assignments are left in the assignment list, the search is over.

5 The hierarchical distributed system

The advantages of the centralized approach are an easy-to-handle topology, direct communication paths, and a global state maintained by the master, which offers very good load balancing capabilities. However, in many real world scenarios the master often is not able to access each node directly. Even, in cases where nodes are accessible, long communication delays from the master to several workers would reduce the performance.

We adopt a hierarchical communication topology based on a tree. Tree hierarchies correspond to the actual structure of multi-domain systems, in which just one node in a domain is accessible from nodes of other domains.

In the following subsections, we describe the management of the logical topology and outline the hierarchical load balancing scheme.

5.1 Topology management

The hierarchy is structured into administration nodes (admins) and worker nodes (workers). The root node represents a special admin.

Worker nodes perform the actual mining task, i.e. they explore a part of the search tree. Furthermore, worker nodes create new subtasks for idle workers by pruning a node of their current search tree. Once the subtask has finished, the result is sent to the parent admin node. Workers are the leaf elements in the communication tree.

Admin nodes manage other nodes and merge the partial results received from their children. Admins represent inner nodes organized in a tree with one or more hierarchical levels. All admins (except root) eventually propagate their aggregated partial results to their parent node.

All computing nodes, except the root, join the system by registering themselves to their parent node. Even if there is no restriction on the hierarchy depth, flat hierarchies have to be preferred in order to avoid long communication paths. However, the branching factor at admin nodes should be limited to avoid a bottleneck similar to the one of the centralized master-workers approach. It should be noticed that, in contrast to the centralized system, the hierarchical topology

can be extended as soon as an admin may represent a bottleneck. A dynamic topology management is out of the scope of this paper.

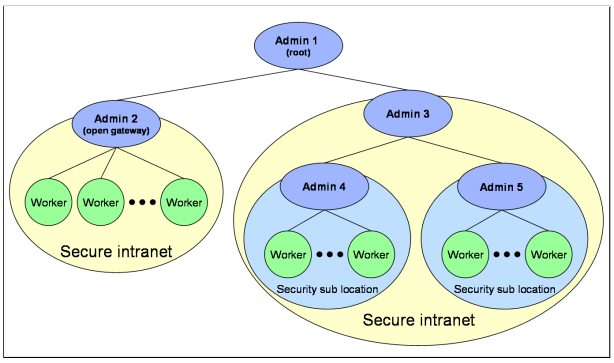


Fig. 2. Schematic example hierarchy

Figure 2 shows an example hierarchy with a root node managing two admins. Admin 2 manages several workers within a secure intranet, which are only accessible from the admin (gateway). Admin 3 is also part of a secure intranet but manages two further security sub-locations.

The next section describes the DLB scheme, which has been designed to work efficiently in hierarchical topologies.

5.2 The load balancing scheme

As in the centralized master/workers approach, the mining task is distributed by partitioning the search space. In this case, the management task is also distributed over several admins at different levels of the hierarchy. Each admin node manages a job pool to serve idle nodes in its subtree. When the job pool size is below the threshold, the admin sends job requests to its child nodes to solicit the generation of new subtasks (search tree pruning). The admin is also logically connected to the rest of the system through its parent node. For a global load distribution the admin has to send job requests to its parent as well, which in turn will forward the request to another branch of the communication topology. However, this naive approach would involve the whole system into the job acquisition process making vain the scalability potential of the hierarchical topology. For the sake of scalability it is necessary to preserve locality in subtrees of the hierarchy. In the next sections we introduce the concept of local and global load balancing that ensure locality in the subtrees.

Local load balancing An admin node performs Local Load Balancing (LLB) to distribute and balance the load among its children (either admins or workers).

From the local point of view, the admin still performs a centralized DLB (section 4). An admin node manages a pool of unprocessed jobs and a threshold J_L is used to trigger job-donation requests to its children. We refer to J_L as the absolute *local* job pool threshold. Children are treated the same way regardless of being workers at a leaf level or further admins. As in the centralized approach, when a worker receives a job-request, it will prune a part of the local search tree according to the *pruning heuristics* (see section 4) and send the new job to the parent node. In case a job request is sent to an admin node, it is forwarded by applying the same LLB policy at each intermediate admin until a worker is reached.

Global load balancing While the LLB activities in a node try to balance the load among its children, the load balancing activities generated by upward requests to its parent node are referred to as Global Load Balancing (GLB). An upward request is triggered by a node when the job pool size is smaller than a relative *global* threshold α_G, $J < J_G = \alpha_G \cdot C$.

Locality is a crucial aspect for the scalability of hierarchical systems. Communication and load balancing activities within subtrees have to be autonomous up to a certain degree to prevent that global communication may limit the scalability. An admin node must first try to satisfy demands for new jobs within its own subtree and, as last extent, it will send a request to its parent node. Therefore, the global threshold is set below the local one ($\alpha_G < \alpha_L$), avoiding global requests when they are not necessary.

Another important aspect in DLB is the donation of a job upon a job-donation request. Donations to child nodes are granted immediately. In contrast, donations to the parent node should not be granted if this may cause starvation of child nodes. To avoid this problem, a third threshold parameter is introduced called the donation threshold α_D, which is related to upward job donation. Thus, donations to the parent node are only granted if $J >= \alpha_D \cdot C = J_D$.

Figure 3 illustrates this threshold in combination with J_L and J_G. The job pool size of Admin 2 is below the local and the global thresholds and thus it generates requests to its children and also to its parent node. The job pool size of Admin 1 is below the local threshold but above the global one. Therefore, it generates downward requests to the child nodes but no upward requests to its parent. If Admin 1 receives a request from its parent node, it would not donate a job from the pool. In general, donating jobs upwards involves more communication than downward donation. To avoid small jobs resulting in frequent GLB, a computing node donating upwards chooses the biggest job from the current job pool (according to the *pruning heuristics*). This strategy reduces global communication and, thus, contributes to the system scalability.

The previous two sections described the hierarchical DLB scheme. Another important aspect in hierarchical systems is the aggregation of information and states. Aggregation makes it possible to approximate a global state for good DLB performance and keep the system scalable anyway. The next section describes this more in detail.

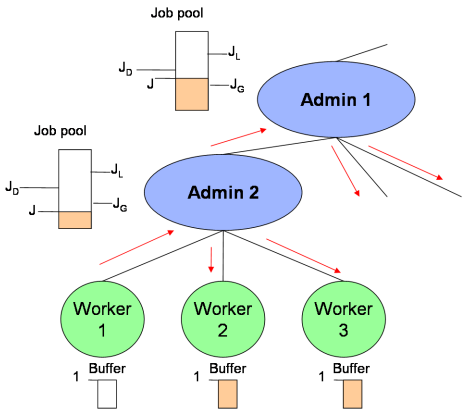


Fig. 3. Hierarchical topology

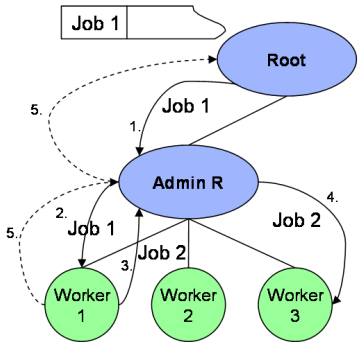


Fig. 4. Pruning Problem

Hierarchical aggregation In centralized systems decisions can be made with a global state ensuring easy and effective load balancing. Hierarchical systems must aggregate system information as it is not possible to maintain the global state in all levels of the hierarchy, i.e. a node should only know about its children and its parent node.

There exist two important problems, which have to be solved. The first one concerns the assignment lists and the second one concerns the termination detection of the hierarchical distributed mining run.

As described in section 4 the efficiency of the donor selection policy is based on the correct knowledge of the global assignment list. However, in the hierarchical system this is not straightforward when several levels are involved, as Figure 4 shows.

In step 1 Root assigns job 1 to its child Admin R and adds the job to its assignment list. Admin R assigns the job to Worker 1, which starts a mining process. The worker prunes a part from job 1 resulting in a new job 2. If job 1 completes before job 2, job 1 is reported as finished to Root (via Admin R) even though a part of the original job is still mined in the subtree. As job 1 will be removed from the assignment list of Root the branch is no longer considered for pruning.

Thus, it is not enough for an admin to remember the job ID and its assignment time. The reason is that job pruning can also occur at deeper levels and thus is not recognized by the admin.

To solve this problem, we adopted a job identifier (job ID) with a hierarchical structure which allows aggregated state information about jobs in a subtree. When a job is propagated downwards the logical topology, at each intermediate node (admin) a digit is appended in a dotted notation (e.g., "3.5.2"). The hierarchy of job IDs respect the hierarchies of tasks in the search tree. In particular, the prefix identifies the parent task and the last appended digit is a sequen-

tial counter that uniquely identifies the subtask. Job completion messages from children are aggregated by means of their ID prefix to detect the completion of the parent task. When this last condition is met, the admin node sends the aggregated job completion message up to its parent.

The hierarchical job ID system ensures that the assignment lists always represent the correct state of jobs in a subtree, thus enabling the donator selection policy to work properly in the whole hierarchy.

The termination detection problem is also solved by the hierarchial ID system. The system ensures that jobs are only reported as completed if all descendant jobs inside the same subtree have also been reported as completed. When the root node detects the completion of the initial job, the whole mining task is over.

6 Experimental results

To show the runtime behavior and the functionality of the hierarchical load balancing scheme we set up different long-distance hierarchical system configurations. A pool of 57 heterogenous computing nodes (1.5GHz / 0.5GB up to 3.4MHz / 1GB) were located at the ICAR-CNR[3] in Italy (24 nodes) and at the University of Konstanz (UniKN) in Germany (33 nodes).

The test configurations are based on two basic scenarios. The first scenario has two worker clusters at different locations (UniKN, ICAR) each managed by an admin, which also resides at the corresponding location. The root node is located at UniKN and manages the admins of both clusters. The second scenario introduces a third worker cluster located at UniKN.

In the hierarchical tests the number of participating workers varies from 12 to 48. The workers are distributed equally among the clusters. Table 1 shows the different configurations of both scenarios.

The hierarchical configurations are compared to the centralized approach. This involves the same computing nodes, which however, are directly managed by the master node.

All tests have been conducted on the well-known and freely available *NCI AIDS Antiviral Screen* screening dataset from the National Cancer Institute (NCI) [1]. A total number of 37171 molecules have been divided into a focus and a complement partition according to an activity threshold (0.5), respectively of 325 compounds and 36846 compounds.

For speedup analysis the sequential algorithm was executed on each of the heterogenous machines with focus support thresholds (see Section 3) of 4%, 5%, 6% and 7%. The runtime of the fastest machine is applied for later speedup analysis. Figure 5 shows the sequential runtime as well as the number of reported discriminative fragments and the number of search tree nodes that have been explored. The graphs depict the exponential problem space. Even though the number of reported fragments is quite low due to the complement threshold

[3] Istituto di Calcolo e Reti ad Alte Prestazioni, Sezione di Palermo, Consiglio Nazionale delle Ricerche, Italy

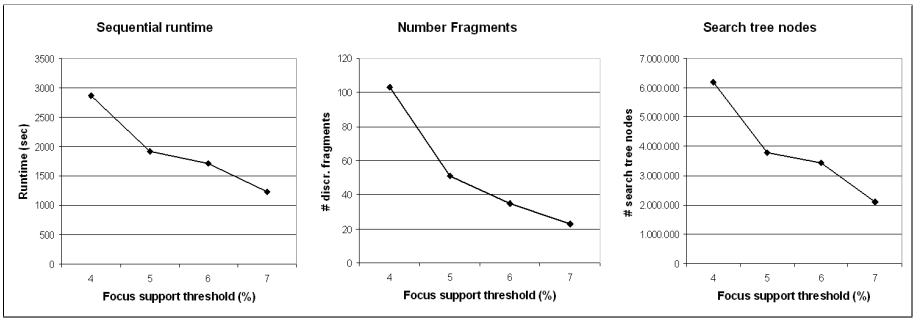


Fig. 5. Sequential performance measures

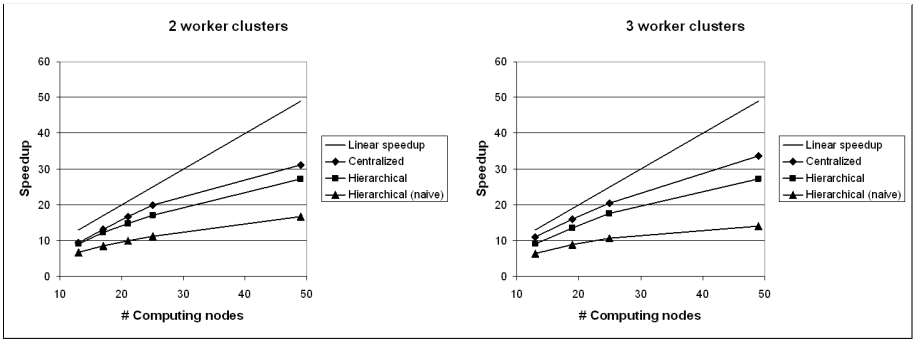


Fig. 6. Speedup comparison

filtering, the number of search tree nodes that have been explored shows a huge search space.

In the distributed tests, the focus support threshold is fixed to a relative value of 4%. For the given focus partition of 325 molecules a fragment must occur in at least 13 of them to be frequent. The complement threshold is set to 0.01%. Therefore, a frequent fragment is also a discriminative fragment if it occurs in a maximum of 4 molecules of the complement partition.

The two graphs of Figure 6 show the speedup values of the hierarchical tests (see Table 6). For each configuration the speedup values are compared to the speedup of the corresponding centralized approach and also to the values of the naive hierarchical approach without the differentiated hierarchical load balancing (thresholds and aggregation).

As expected, the speedup of the hierarchical system is below the one of the centralized system. This is due to the global state maintained by the centralized approach, which provides better DLB quality. Nevertheless, the performance of the hierarchical approach is still very good. In both scenarios (2 and 3 clusters) the naive approach performs poorly due to the missing job-state aggregation

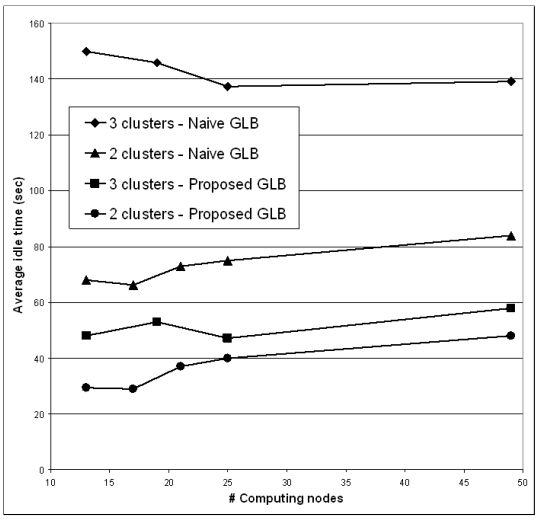


Fig. 7. Average idle time

Basic scenario	Worker UniKN	Worker ICAR	All nodes
2 clusters	5	5	13
	7	7	17
	9	9	21
	11	11	25
	23	23	49
3 clusters	2x3	3	13
	2x5	5	19
	2x7	7	25
	2x15	15	49

Table 1. Test configurations

(bad donor selection) and the unrestricted parent requests (no DLB thresholds differentiation), which results in unnecessary job prunings. As a result the idle times increase. The graph in Figure 7 shows the average idle time for the naive and proposed hierarchical DLB approaches.

The average idle time increases when number of clusters increases, i.e. with a more distributed system. As expected, the naive version shows longer idle times. Note the bigger difference between the idle time in the 2 and 3 clusters configuration of the naive version with respect to the proposed one. If the hierarchy becomes more complex, the impact of insufficient aggregated state information and undifferentiated load policies becomes more relevant. The experimental results have provided evidence of the effectiveness of the proposed hierarchical DLB scheme applied to multi-domain environments.

7 Conclusions

In this paper we have presented a distributed approach to the frequent subgraph mining problem for computational environments that are characterized by a hierarchical communication topology. The system widens the architectural bottleneck of centralized and peer-to-peer systems, which cannot operate in multi-domain environments due to security restrictions. The proposed approach was successfully applied to the discriminative molecular fragments mining problem, but can also be applied to many problems based on a search tree in which the search space is unknown in advance.

The system applies a sophisticated hierarchical aggregation as well as a differentiated DLB scheme to reduce the drawback of missing global state information

(as opposed to centralized approaches). The system maintains locality within the hierarchical structure to keep it scalable.

The experimental results show that the hierarchical approach performs close to the centralized one and is able to exploit potential computing power not eventually accessible.

Future work will deal with a topology management that dynamically and efficiently organizes participating nodes by taking into account both security policies and network delays.

8 Acknowledgements

This work was partially supported by the DFG Research Training Group GK-1042 "Explorative Analysis and Visualization of large Information Spaces". We also thank Christian Stolze of the University of Konstanz, Germany and Pietro Storniolo of ICAR-CNR, Italy for their administrative support of the computing facilities.

References

1. http://dtp.nci.nih.gov/docs/aids/aids_data.html.
2. K. Antonis, J. Garofalakis, I. Mourtos, and P. Spirakis. A hierarchical adaptive distributed algorithm for load balancing. *Journal of Parallel and Distributed Computing*, 64:151–162, 2004.
3. C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. IEEE International Conference on Data Mining (ICDM 2002, Maebashi, Japan). pages 51–58, December 09-12, 2002.
4. S. Dandamudi and K. Lo. A Hierarchical Load Sharing Policy for Distributed Systems. *5th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, Haifa, ISRAEL*, 1997.
5. G. Di Fatta and M. R. Berthold. Distributed mining of molecular fragments. *Proc. of IEEE DMGrid, Workshop on Data Mining and Grid of IEEE ICDM*, 2004.
6. G. Di Fatta and M. R. Berthold. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on High Performance Computational Biology*, 17(8), August 2006.
7. V. Kumar, A. Grama, and V. N. Rao. Scalable load balancing techniques for parallel computer. *Journal of Parallel and Distributed Computing*, 22(1):60–79, July 1994.
8. R. Pollak. A hierarchical load balancing environment for parallel and distributed supercomputer, International Symposium on Parallel and Distributed Supercomputing, Fukuoka, Japan. 1995.
9. T. Washio and H. Motoda. State of the art of graph-based data mining. *ACM SIGKDD Explorations Newsletter*, 5(1):59–68, July 2003.
10. Y. Xu, T. K. Ralphs, L. Ladanyi, and M. J. Saltzman. Alps: A framework for implementing parallel search algorithms, The Proceedings of the Ninth INFORMS Computing Society Conference. 2004.

Customizing the APPSPACK Software for Parallel Parameter Tuning of a Hybrid Parallel Support Vector Machine

Tatjana Eitrich[1], Bruno Lang[2], and Achim Streit[1]

[1] Central Institute for Applied Mathematics, Research Centre Jülich, Germany
[2] Applied Computer Science and Scientific Computing Group, Department of Mathematics, University of Wuppertal, Germany

Abstract. We consider the problem of tuning parameters for the learning method support vector machine. The APPSPACK software, an asynchronous parameter tuning method, is well suited for SVM parameter fitting due to several characteristics. No derivative information is needed for bound constrained optimization and the code can be run in parallel mode. Recently, a hybrid parallel support vector machine has been proposed. To couple both parallel packages, the APPSPACK software needs to be customized to allow for a parallel function evaluation in addition to the parallelism provided by APPSPACK. In this paper we describe our customization of the APPSPACK software to facilitate a top down parallelism in SVM parameter tuning.

1 Introduction

The support vector machine (SVM) is a well-known and widely accepted method of supervised learning [1]. This classification method involves a parameter tuning phase, in which the validation data are used to adjust a set of learning parameters. Usually two parameters are tuned in SVM learning – the parameter of the so called kernel function and the weight for training error penalization. We have shown [2], that extended learning models with additional parameters for cost sensitive learning enhance performance of classification. Tuning of more than two parameters is very time consuming and requires sophisticated methods such as genetic algorithms [3]. The usual grid search techniques [4], that scan the whole parameter space using predefined intervals for test cases, go beyond the scope of parameter tuning in a complex model due to their enormous computational overhead. In [5] we have tested the freely available APPSPACK software package [6] for tuning the SVM learning parameters. APPSPACK is derivative free optimization for unconstrained and bound constrained optimization problems. As agent between the SVM method and APPSPACK we implemented a cost sensitive quality function that measures the performance a certain parameter bundle gives for a data set [2]. Our tests with the serial support vector machine gave promising results. Recently, we have parallelized the SVM with a hybrid parallel scheme for usage on SMP (shared memory processor) clusters [7].

The algorithm heavily reduces training and validation time. In our opinion, the coupling of parallel APPSPACK software and parallel SVM software would improve the parameter optimization process due to the following fact. By using the same number of CPUs, the number of parallel workers in APPSPACK decreases, at the same time each SVM validation needs less time. The overall benefit comes from the fact, that each function evaluation is faster, APPSPACK receives the results in shorter intervals and thus is able to better scan the parameter space by choosing promising search directions. In addition, we assume better load balance for the workers. So far, the coupled usage was not possible. APPSPACK calls external function evaluations. Such a call to a parallel executable would be trapped by the scheduler of the parallel machine we use. There, it remains until new resources are available. In this paper we describe the way we have customized the APPSPACK software to avoid this problem.

The remainder of this paper is structured as follows. In Section 2 we review the basics concepts of support vector machine learning and the parameter tuning problem. In Sect. 3 we briefly describe the APPSPACK software and motivate the reasons for choosing this optimization package for SVM parameter tuning. In Sect. 4 we introduce the parallelism of our support vector machine. A detailed description of the software customization including the new parallel scheme is given in Sect. 5. Test results and a discussion of the new software package are given in Sect. 6. In Sect. 7 we summarize our findings and show directions to future work.

2 Support Vector Machine and Parameter Tuning

The support vector machine is state-of-the-art machine learning method for supervised classification and regression [8]. Classification is one of the most important data mining tasks in our days. Given a training set

$$\left\{\left(\boldsymbol{x}^i, y_i\right) \in \mathbb{R}^n \times \{-1, 1\}, \quad i = 1, \ldots, l\right\} \ ,$$

where $l \in \mathbb{N}$ is the number of instances and $n \in \mathbb{N}$ the number of attributes in the data set, the task of support vector machine learning is to find a hypothesis function $h : \mathbb{R}^n \to \mathbb{R}$ that can be used to classify unseen data. The hypothesis function, the sign of which is used to classify a new point $\boldsymbol{x}$, is of the form

$$h(\boldsymbol{x}) = \sum_{i:\alpha_i > 0} y_i \alpha_i K(\boldsymbol{x}^i, \boldsymbol{x}) + b^* .$$

It is mainly controlled by the so-called Lagrange multipliers α_i $(i = 1, \ldots, l)$. They can be determined via the solution of the quadratic programming (qp) problem

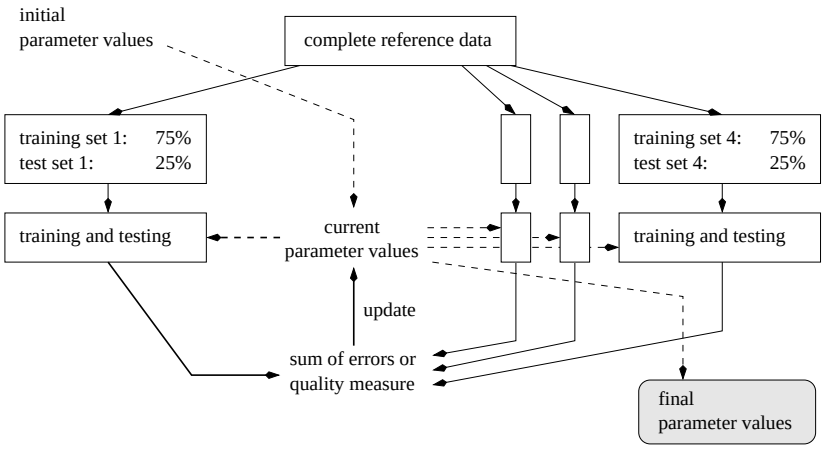


Fig. 1. Structure of parameter tuning with a 4-fold cross validation method.

$$\begin{aligned} &\min_{\boldsymbol{\alpha}\in\mathbb{R}^l} \quad \frac{1}{2}\sum_{i,j=1}^{l} y_i y_j \alpha_i \alpha_j K(\boldsymbol{x}^i, \boldsymbol{x}^j) - \sum_{i=1}^{l} \alpha_i \\ &\text{s.t.} \\ &\sum_{i=1}^{l} y_i \alpha_i = 0 \ , \\ &0 \le \alpha_i \le C \quad (1 \le i \le l) \ . \end{aligned} \tag{1}$$

The function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is known as the kernel [1] and measures similarity between input vectors [9]. $C \in \mathbb{R}_+$ is an SVM internal error penalization parameter which controls the trade-off between a large margin and the corresponding training errors. We refer to [1] for a detailed description of the SVM learning problem. One of the main challenges when using SVM-based methods is parameter selection. Several data dependent parameter values need to be adjusted [2]. Different methods for tuning the parameters have been proposed [10]. One of them is a search procedure that iteratively creates new parameter values using quality results from k-fold cross validation. In Fig. 1 we explain this method for $k = 4$. A k-fold cross validation includes k SVM training and test stages as well as a final combination of the results to obtain a quality measure value. We are working with our implementation of the decomposition method which includes the fast projection method proposed in [11]. However, a single SVM training is expensive for large data. Thus, a complete validation takes long time. Parameter tuning usually means to perform a large number of validation stages. Efficient and fast methods are of great interest since they allow for an extensive scan of the parameter space and usage of additional parameters, e.g. for sensitive classification of highly unbalanced data [2]. In Sect. 4 we shortly discuss our parallelized SVM validation and training software. Our SVM para-

meter tuning is based on the evaluation of a cross validation quality measure. Given a parameter vector $\boldsymbol{p} \in \mathbb{R}^m$, where $m \in \mathbb{N}$ is the number of parameters to be tuned, the quality measure is simply a function f, that returns a value $f(\boldsymbol{p})$. Usually, a quality measure is defined in $[0, 1]$, so does our smooth E-measure [5]

$$E_\beta(\boldsymbol{p}) := 1 - \frac{\left(\beta^2 + 1\right) \mathrm{pr}(\boldsymbol{p}) \cdot \mathrm{se}(\boldsymbol{p})}{\beta^2 \cdot \mathrm{pr}(\boldsymbol{p}) + \mathrm{se}(\boldsymbol{p})} \ . \tag{2}$$

Sensitivity (se) and precision (pr) are computed using a smooth error measure [2]. β is an additional parameter to tune the influence of sensitivity. The default value is $\beta = 1$. Thus, the parameter tuning task is to minimize the function (2) with $2 \cdot m$ bound constraints for the parameter values. Usually, the lower bound vector is $\mathbf{0}$ and the upper bound is $\boldsymbol{\infty}$.

3 Description of the Parallel APPSPACK Software

APPSPACK [6] is software for solving unconstrained and bound constrained optimization problems. It implements asynchronous parallel pattern search [12], a method in the class of direct search methods. APPSPACK has been designed for problems characterized by expensive function evaluations. APPSPACK has the following advantages [13]:

- No derivative information is required during optimization.
- The procedure for evaluating the objective function can be executed via a separate program.
- The software is freely available.
- The code can be run in serial and parallel.
- Up to 100 parameters may be tuned at the same time.
- Bound constraints are permitted on a dimension wise basis.

The user needs to provide the function to be optimized. The following features need or may be stated:

- the name of the executable to be used,
- the number of parameters to be tuned,
- initial point,
- lower bounds,
- upper bounds,
- scaling,
- step length for stopping condition,
- maximal number of optimization steps,
- digits of precision in output,
- ...

APPSPACK offers a parallel MPI-based mode that implements a so called master-worker approach, where all processors but one are workers. In this MPI mode the master processor owns the executor and assigns points to w workers

for evaluation. APPSPACK executes multiple function evaluations in parallel. MPI messages containing points for evaluation are sent to idle workers which are then marked as busy. Each worker calls its own evaluator. Later, the result of the function evaluation is sent to the master, see Fig. 2. Since the master controls the optimization, it is not necessary to inform workers about constraints and other restrictions. Workers receive parameter bundles and use them for evaluation. APPSPACK is able to handle error messages. In case a function evaluation fails, the optimization can proceed. The evaluator works as follows. A function

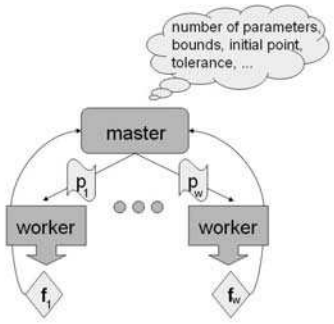


Fig. 2. Master-worker scheme of the APPSPACK software.

input file containing the point to be evaluated is created. An external system call is made to the user-provided executable that calculates the function value. After the function value has been computed, the evaluator reads the result from the function output file. Then, input and output files are deleted. To prevent the parallel processes from overwriting information, each call includes uniquely named files using tag numbers for each point. APPSPACK works in a so called asynchronous mode. The asynchrony comes about as a consequence of the fact that the search along a particular direction continues without waiting for searches along other directions to finish. The user may customize APPSPACK. In general, two components are of interest. In case the user is interested in directly computing the function, a new evaluator without the external call needs to be created. The manager-worker relationship and the parallelism of APPSPACK can be modified via a new executor. In Sect. 5 we describe the way we customized the APPSPACK executor to allow for multi-level parallelism.

4 Hybrid Parallel Support Vector Machine

In [7] we have presented a parallel support vector machine training method for shared memory systems. The SVM training, i.e. the solution of the quadratic program (1), suffers from large data sets [14]. Since data sets are becoming

increasingly large in various fields of research, e.g. in text mining, parallel SVM training is essential to improve performance. Our parallel SVM algorithm is based on library and loop parallelism. Calls to the ESSLSMP library (Engineering Scientific Subroutine Library for Shared Memory Parallel Machines) [15] as well as OpenMP loop level parallelism lead to a scalable training method. Our code has been tested on the JUMP supercomputer [16]. For details to the serial algorithm and the parallelization we refer to [7]. In [17] a distributed k-fold cross validation for support vector machine learning has been proposed. The independent training and test stages of a complete cross validation loop are assigned to different processors. Their number should fit into the validation model. Usually we perform 8-fold cross validation, so that c=2, c=4 or c=8 processors may be used to achieve good speedup values. The root process collects validation results by using MPI reduction operations and computes the quality measure value. Both parallel methods have been merged into a hybrid parallel SVM validation [17]. The parallel scheme is top down. The outer parallel routine implements distributed validation, where each step of the validation includes a shared memory parallel SVM training. Each CPU should perform at least one validation step ($c \leq k$). Therefore the scalability of parallel validation is limited to k. Remaining resources may be used for the parallel training. The scheme is given in Fig. 3. Our hybrid parallel SVM validation method shows promising

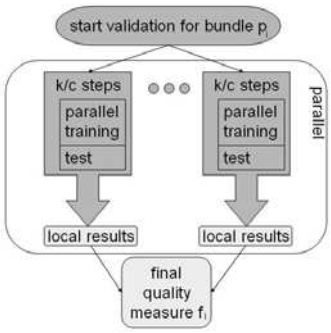


Fig. 3. Hybrid parallel cross validation method.

performance on the SMP cluster JUMP. However, the attainable speedup is limited depending on the following characteristics:

- The shared memory parallelism is useful for training sets with more than 2000 points only.
- The speedup of the training is limited to approx. 8 [17], which is due to some serial parts of the training and the fact, that the maximal size of the working set (SVM internal size for optimization, size for ESSL routines) is limited to the available memory.

– The number of CPUs used for distributed validation is limited to k, which in our settings is at most 8.

Therefore, on SMP machines with hundreds of CPUs, a third level of parallelism is desirable. We can increase parallelism by using the parallel APPSPACK software for parameter optimization. In the following section we describe the way we customized both software packages to allow for a coupled usage.

5 Customizing both Software Packages

In [5] the parallel APPSPACK software was successfully applied for SVM parameter tuning. There, the serial support vector machine was used. While the support vector machine is implemented in Fortran, APPSPACK is written in C++. The external function evaluation via a call to the SVM executable was easy to manage. To allow for usage of the hybrid parallel support vector machine, a customized version needs to be implemented. The new scheme needs to fulfill the following requirements.

– The external call is no longer possible. Both codes have to be merged together in a single executable.
– The C++ software is on top and provides the main program.
– The parallelsim is no longer limited to a master and its workers. Each worker owns its private dummy processors as well.
– The master is not aware of the dummies and communicates with the workers exclusively.

The new scheme is given in Fig. 4. Please note, that each worker also acts as a dummy during the validation.

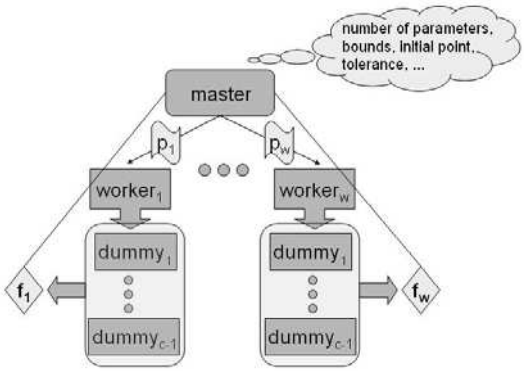


Fig. 4. New scheme of customized APPSPACK and SVM software.

5.1 APPSPACK

Customizing APPSPACK to allow for usage of our parallel SVM code consists of several important steps. First, the external function call needs to be replaced by a direct function call and second, MPI is to be used directly rather than the APPSPACK::GCI interface to MPI. Both settings are provided in a special custom executor example that comes with the APPSPACK source code in the latest version of APPSPACK (4.0.2, released december 2005). We modified and customized this example to allow for the desired parallelism and a call to the Fortran library rather than an internal C++ function. The most important changes are given here.

main.cpp In the main file we built new MPI communicators. So far, the default MPI communicator has been used, because all processors are either master or worker. All in all we need two new schemes for communication. A row communicator (row_comm) is required for communication between the master and all workers. The real dummies are not included in this communicator. A column communicator (col_comm) is essential for communication between each worker and its dummies. The master is not a part of this communicator. Therefore, the main file includes the following additional lines:

```
int rows;     //number of dummies per worker
int cols;     //number of workers
int members;  //master and workers together
int row;
int row_comm; //new communicator
int number;   //master:0, worker:1,w
int col;
int col_comm; //new communicator
int dum;      //worker:0,dummies:1,c-1
rows=c;
cols=w;
members=cols+1;
//create row communicator
if (rank<members){
  row=0;}
else{
  row=MPI_UNDEFINED;}
MPI_Comm_split(MPI_COMM_WORLD,row,rank,&row_comm);
//rank is the number in MPI_COMM_WORLD
number=-1;
if (rank<members) MPI_Comm_rank(row_comm,&number);
//create column communicator
if (rank>0){
  col=(rank-1)%cols;}
else{
  col=MPI_UNDEFINED;}
MPI_Comm_split(MPI_COMM_WORLD,col,rank,&col_comm);
dum=-1;
if (rank>0) MPI_Comm_rank(col_comm,&dum);
```

master.cpp In the master routine the row communicator is used to send messages to the workers.

```
  for (int i=1;i<members;i++) //problem size
    MPI_Send(&n,1,MPI_INT,i,SIZE,row_comm);
[...]
  for (int i=1;i<members;i++) //quit message
    MPI_Send(&n,1,MPI_INT,i,QUIT,row_comm);
```

executor.cpp The custom executor object was modified. We added new arguments to the executer and to the private variables. The executor now sends trial points to the workers, not to dummies.

```
  MPI_Send(&tag_in,1,MPI_INT,idx,XTAG,row_comm);
  MPI_Send(x,n,MPI_DOUBLE,idx,XVEC,row_comm);
[...]
  MPI_Iprobe(MPI_ANY_SOURCE,XTAG,row_comm,&flag,&mpiStatus);
[...]
  MPI_Recv(&tag,1,MPI_INT,source,XTAG,row_comm,&mpiStatus);
  MPI_Recv(&f,1,MPI_DOUBLE,source,FVAL,row_comm,&mpiStatus);
```

worker.cpp In the worker routine the function evaluation was replaced by a call to the Fortran SVM library.

```
#define feval __mainx_NMOD_feval //Link to F90 source

extern "C" double feval(const int *n,const double *x,const int *col_comm,
const int *rank,const int *dum,const int *procs,const int *rows);
```

The workers in row_comm receive messages from the master and the executor. Each worker forwards messages to its dummies by using the column communicator. The while-loop was modified to avoid a break of a worker without its dummies. Therefore, a new stopping variable was defined, that informs dummies before the break.

```
  int stop;
  if (number>0)
    MPI_Recv(&n,1,MPI_INT,0,SIZE,row_comm,&status);
  MPI_Bcast(&n,1,MPI_INT,0,col_comm);
[...]
  while (1){
    if (number>0){
      MPI_Probe(0,MPI_ANY_TAG,row_comm,&status);
      msgtag=status.MPI_TAG;
      if (msgtag==QUIT) stop=1;
      for (int i=1;i<rows;i++) //stop message
        MPI_Send(&stop,1,MPI_INT,i,0,col_comm);
      if (msgtag==QUIT) break;}
    if (dum>0){
      MPI_Recv(&stop,1,MPI_INT,0,0,col_comm,&status);
```

```
      if (stop==1) break;}
    if (number>0){
      MPI_Recv(&tag,1,MPI_INT,0,XTAG,row_comm,&status);
      MPI_Recv(x,n,MPI_DOUBLE,0,XVEC,row_comm,&status);}
    //send point to other dummies
    MPI_Bcast(x,n,MPI_DOUBLE,0,col_comm);
[...]
    f=f_eval(&n,x,&col_comm,&rank,&dum,&procs,&rows);
    if (number>0){
      //send result to executor
      MPI_Send(&tag,1,MPI_INT,0,XTAG,row_comm);
      MPI_Send(&f,1,MPI_DOUBLE,0,FVAL,row_comm);}}
```

solver.cpp The solver and related files do not use the parallelism of APPSPACK. This is due to the master-worker approach. The master iteratively generates trial points, that are assigned to idle workers. The master exclusively uses the APPSPACK solver class. No single worker enters it. Therefore, we need not to modify the solver class and the methods, used in this solver.

5.2 SVM

The function evaluation cannot be realized via an external call. Thus, we modified the Fortran software and built a library out of it. Our main program was replaced by a function

```
f(n,p,comm,rank,dum,procs,rows)
```

where n is the number of parameters, p the current parameter vector, dum the position in the column, rows the number of dummies in the column. Here, the worker is a dummy, too. The values are taken from the function call in the C++ worker routine. After receiving the values, the SVM validation routine, that performs parallel cross validation, is called. In the Fortran software the new column communicator col_comm is used instead of the default MPI communicator. Thus, each worker performs validation always together with the same dummies. After the validation, each worker collects local results and computes the quality measure f, which is then sent to the master for evaluation.

5.3 Command Line and Initialization

In the usual APPSPACK software an .apps file needs to be specified, where the solver information is given. In the customized example, a communication via .apps file is not included. Instead, the master routine may be used to provide the master with information. As we mentioned earlier, the workers need not to know the details. They receive messages with trial points and call the SVM routine. In the master routine we specify the initial point, bounds, number of evaluations, the debug level and other parameters. In the commanline, the number of parameters to be tuned in given. We extended the commandlline by the options

```
-cols <int>
-rows <int>
```

to enable a flexible usage of the parallelism we introduced here.

```
//read number of workers
sscanf(argv[3],"%d",&w);
//read number of dummies for validation
sscanf(argv[5],"%d",&c);
```

We plan to add more arguments to the APPSPACK commandline, such as the dataset name and the parameter bounds to omit changes in the source code.

6 Tests

We present results obtained in tests with our new software. As a first example we have chosen the well-known breast cancer dataset from the University of Wisconsin Hospitals, Madison [18]. It includes 699 points, and each instance bears one of two possible class labels, benign or malignant. The number of malignant reference points is 241. From the ten attributes we removed the first one, since it codes the sample number and does not contribute relevant information. Of the 699 points in the dataset, we used 550 for 8-fold cross validation. We have tested the following three settings:

	test 1	test 2	test 3
master	1	1	1
number of workers	12	6	3
additional dummies per worker	0	1	3
number of CPUS	13	13	13

Our intent is to show that by using a constant number of CPUs, the parallel scheme may help to reach the solution faster. Please note, that we do not examine speedup for varying number of CPUs. For the SVM this was done in [17]. In all tests the same solution was reached. We analyze the time that was spent for optimization as well as the work load of the workers. Our time measurements were performed by using the MPI_Wtime() function for the master. The times were

	test 1	test 2	test 3
time in seconds	28.2	16.3	12.4
evaluated points	89	84	68

Our results show that by using our new parallel scheme both, the time and the number of evaluated points, decreased. It is interesting to study the load of the workers. Load is given by the number of function evaluations (full SVM validations), since every SVM validation consumes approximately the same amount of time.

	test 1	test 2	test 3
worker 0	18	18	23
worker 1	17	19	23

```
worker 2      15      19      22
worker 3      17      17
worker 4       6       6
worker 5       5       5
worker 6       3
worker 7       3
worker 8       2
worker 9       1
worker 10      1
worker 11      1
```

We conclude that our assumptions stated in Sect.1 are satisfied for this data set. Our tests with this small example need to be continued using larger data sets.

7 Concluding Remarks and Future Work

We have shown, how the parallel APPSPACK optimization software may be coupled with our recently proposed hybrid parallel software. We customized both codes to allow for top-down parallelism between them. The main advantage of this coupling is the possibility to use parallel parameter tuning, where each step itself is processed faster than in the case of a serial function for tuning. In a toy example we verified our assumptions. Further analysis will be on the question how to split the available CPUs into APPSPACK workers, SVM dummies and SVM inner threads efficiently.

Acknowledgement

We would like to thank Tamara Kolda for helpful suggestions concerning the APPSPACK customization and for providing the pre-customized examples within the source code.

References

1. Cristianini, N., Shawe-Taylor, J.: An introduction to support vector machines and other kernel-based learning methods. Cambridge University Press, Cambridge, UK (2000)
2. Eitrich, T., Lang, B.: Efficient optimization of support vector machine learning parameters for unbalanced datasets. Journal of Computational and Applied Mathematics **196** (2006) 425–436
3. Angeline, P.J.: Evolutionary algorithms and emergent intelligence. PhD thesis, Ohio State University (1993)
4. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. (2001) Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.
5. Eitrich, T., Lang, B.: Parallel tuning of support vector machine learning parameters for large and unbalanced data sets. In Berthold, M.R., Glen, R., Diederichs, K., Kohlbacher, O., Fischer, I., eds.: Computational Life Sciences, First International Symposium (CompLife 2005), Konstanz, Germany. Volume 3695 of Lecture Notes in Computer Science., Springer (2005) 253–264

6. Gray, G.A., Kolda, T.G.: APPSPACK 4.0: asynchronous parallel pattern search for derivative-free optimization. Sandia Report SAND2004-6391, Sandia National Laboratories, Livermore, CA (2004)
7. Eitrich, T., Lang, B.: Data mining with parallel support vector machines for classification. In: Proceedings of ADVIS (to appear). (2006)
8. Abe, S.: Support vector machines for pattern classification. Springer (2005)
9. Schölkopf, B.: The kernel trick for distances. In: Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA, MIT Press (2001) 301–307
10. Chapelle, O., Vapnik, V.N., Bousquet, O., Mukherjee, S.: Choosing multiple parameters for support vector machines. Machine Learning **46**(1) (2002) 131–159
11. Serafini, T., Zanghirati, G., Zanni, L.: Gradient projection methods for quadratic programs and applications in training support vector machines. Optimization Methods and Software **20**(2-3) (2005) 353–378
12. Kolda, T.G.: Revisiting asynchronous parallel pattern search for nonlinear optimization. Technical Report SAND2004-8055, Sandia National Laboratories, Livermore, CA 94551 (2004)
13. Gray, G.A., Kolda, T.G.: Algorithm 8xx: APPSPACK 4.0: asynchronous parallel pattern search for derivative-free optimization. ACM Transactions on Mathematical Software **32**(3) (2006) in press.
14. Graf, H.P., Cosatto, E., Bottou, L., Dourdanovic, I., Vapnik, V.: Parallel support vector machines: the cascade SVM. In Saul, L.K., Weiss, Y., Bottou, L., eds.: Advances in Neural Information Processing Systems 17. MIT Press, Cambridge, MA (2005) 521–528
15. IBM: (ESSL - Engineering and Scientific Subroutine Library for AIX version 4.1)
16. Detert, U.: Introduction to the JUMP architecture. (2004)
17. Eitrich, T., Frings, W., Lang, B.: HyParSVM – a new hybrid parallel software for support vector machine learning on SMP clusters. In Nagel, W.E., Walter, W.V., Lehner, W., eds.: Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference. Volume 4128 of LNCS., Springer (2006) 350–359
18. Mangasarian, O.L., Wolberg, W.H.: Cancer diagnosis via linear programming. SIAM News **23** (1990) 1–18

Parallel Training of CRFs: A Practical Approach to Build Large-Scale Prediction Models for Sequence Data

H.X. Phan[1], M.L. Nguyen[1], S. Horiguchi[2], Y. Inoguchi[1], and B.T. Ho[1]

[1] Japan Advanced Institute of Science and Technology
1–1, Asahidai, Tatsunokuchi, Ishikawa, 923–1211, Japan
{hieuxuan, nguyenml, inoguchi, bao}@jaist.ac.jp
[2] Tohoku University
Aoba 6–3–09 Sendai, 980–8579, Japan
susumu@ecei.tohoku.ac.jp

Abstract. Conditional random fields (CRFs) have been successfully applied to various applications of predicting and labeling structured data, such as natural language tagging & parsing, image segmentation & object recognition, and protein secondary structure prediction. The key advantages of CRFs are the ability to encode a variety of overlapping, non-independent features from empirical data as well as the capability of reaching the global normalization and optimization. However, estimating parameters for CRFs is very time-consuming due to an intensive forward-backward computation needed to estimate the likelihood function and its gradient during training. This paper presents a high-performance training of CRFs on massively parallel processing systems that allows us to handle huge datasets with hundreds of thousand data sequences and millions of features. We performed the experiments on an important natural language processing task (phrase chunking) on large-scale corpora and achieved significant results in terms of both the reduction of computational time and the improvement of prediction accuracy.

1 Introduction

CRF, a conditionally trained Markov random field model, together with its variants have been successfully applied to various applications of predicting and labeling structured data, such as information extraction [1, 2], natural language tagging & parsing [3, 4], pattern recognition & computer vision [5, 7, 6, 8], and protein secondary structure prediction [9, 10]. The key advantages of CRFs are the ability to encode a variety of overlapping, non-independent features from empirical data as well as the capability of reaching the global normalization and optimization.

However, training CRFs, i.e., estimating parameters for CRF models, is very expensive due to a heavy forward-backward computation needed to estimate the likelihood function and its gradient during the training process. The computational time of CRFs is even larger when they are trained on large-scale datasets or using higher-order Markov dependencies among states. Thus, most previous work either evaluated CRFs on moderate datasets or used the first-order Markov

CRFs (i.e., the simplest configuration in which the current state only depends on one previous state). Obviously, this difficulty prevents us to explore the limit of the prediction power of high-order Markov CRFs as well as to deal with large-scale structured prediction problems.

In this paper, we present a high-performance training of CRFs on massively parallel processing systems that allows to handle huge datasets with hundreds of thousand data sequences and millions of features. Our major motivation behind this work is threefold:

• Today, (semi-)structured data (e.g., text, image, video, protein sequences) can be easily gathered from different sources, such as online documents, sensors, cameras, and biological experiments & medical tests. Thus, the need for analyzing, e.g., segmentation and prediction, those kinds of data is increasing rapidly. Building high-performance prediction models on distributed processing systems is an appropriate strategy to deal with such huge real-world datasets.

• CRF has been known as a powerful probabilistic graphical model, and already applied successfully to many learning tasks. However, there is no thoroughly empirical study on this model on large datasets to confirm its actual limit of learning capability. Our work also aims at exploring this limit in the viewpoint of empirical evaluation.

• Also, we expect to examine the extent to which CRFs with the global normalization and optimization could do better than other classifiers when performing structured prediction on large-scale datasets. And from that we want to determine whether or not the prediction accuracy of CRFs should compensate its large computational cost.

The rest of the paper is organized as follows. Section 2 gives the background of CRFs. Section 3 presents the parallel training of CRFs. Section 4 presents the empirical evaluation. And some conclusions are given in Section 5.

2 Conditional Random Fields

The task of predicting a label sequence to an observation sequence arises in many fields, including bioinformatics, computational linguistics, and speech recognition. For example, consider the natural language processing task of predicting the part-of-speech (POS) tag sequence for an input text sentence as follows: "*Rolls-Royce*_NNP *Motor*_NNP *Cars*_NNPS *Inc.*_NNP *said*_VBD *it*_PRP *expects*_VBZ *its*_PRP$ *U.S.*_NNP *sales*_NNS *to*_TO *remain*_VB *steady*_JJ *at*_IN *about*_IN *1,200*_CD *cars*_NNS *in*_IN *1990*_CD ._."

Here, "*Rolls-Royce Motor Cars Inc. said* ..." and "NNP NNP NNPS NNP VBD ..." can be seen as the input data observation sequence and the output label sequence, respectively. The problem of labeling sequence data is to predict the most likely label sequence of an input data observation sequence. CRFs [11] was deliberately designed to deal with such kind of problem.

Let $\mathbf{o} = (o_1, \ldots, o_T)$ be some input data observation sequence. Let $\mathcal{S}$ be a finite set of states, each associated with a label l ($\in \mathcal{L} = \{l_1, \ldots, l_Q\}$). Let $\mathbf{s} = (s_1, \ldots, s_T)$ be some state sequence. CRFs are defined as the conditional probability of a state sequence given an observation sequence as,

$$p_\theta(\mathbf{s}|\mathbf{o}) = \frac{1}{Z(\mathbf{o})}\exp\left(\sum_{t=1}^{T} \mathbf{F}(\mathbf{s}, \mathbf{o}, t)\right), \tag{1}$$

where $Z(\mathbf{o}) = \sum_{\mathbf{s'}} \exp\left(\sum_{t=1}^{T} \mathbf{F}(\mathbf{s'}, \mathbf{o}, t)\right)$ is a normalization factor summing over all label sequences. $\mathbf{F}(\mathbf{s}, \mathbf{o}, t)$ is the sum of CRF features at time position t,

$$\mathbf{F}(\mathbf{s}, \mathbf{o}, t) = \sum_i \lambda_i f_i(s_{t-1}, s_t) + \sum_j \lambda_j g_j(\mathbf{o}, s_t) \tag{2}$$

where f_i and g_j are *edge* and *state* features, respectively; λ_i and λ_j are the feature weights associated with f_i and f_j. Edge and state features are defined as binary functions as follows,

$$f_i(s_{t-1}, s_t) \equiv [s_{t-1} = l^{'}][s_t = l]$$
$$g_j(\mathbf{o}, s_t) \equiv [x_j(\mathbf{o}, t)][s_t = l]$$

where $[s_t = l]$ equals 1 if the label associated with state s_t is l, and 0 otherwise (the same for $[s_{t-1} = l^{'}]$). $x_i(\mathbf{o}, t)$ is a logical context predicate that indicates whether the observation sequence $\mathbf{o}$ (at time t) holds a particular property. $[x_i(\mathbf{o}, t)]$ is equal to 1 if $x_i(\mathbf{o}, t)$ is *true*, and 0 otherwise. Intuitively, an edge feature encodes a sequential dependency or causal relationship between two consecutive states, e.g., "the label of the previous word is JJ (adjective) and the label of the current word is NN (noun)". And, a state feature indicates how a particular property of the data observation influences the prediction of the label, e.g., "the current word ends with *-tion* and its label is NN (noun)".

2.1 Inference in Conditional Random Fields

Inference in CRFs is to find the most likely state sequence $\mathbf{s}^*$ given the input observation sequence $\mathbf{o}$,

$$\mathbf{s}^* = \text{argmax}_{\mathbf{S}}\ p_\theta(\mathbf{s}|\mathbf{o}) = \text{argmax}_{\mathbf{S}} \left\{\exp\left(\sum_{t=1}^{T} \mathbf{F}(\mathbf{s}, \mathbf{o}, t)\right)\right\} \tag{3}$$

In order to find $\mathbf{s}^*$, one can apply a dynamic programming technique with a slightly modified version of the original Viterbi algorithm for HMMs [12]. To avoid an exponential–time search over all possible settings of $\mathbf{s}$, Viterbi stores the probability of the most likely path up to time t which accounts for the first t observations and ends in state s_i. We denote this probability to be $\varphi_t(s_i)$ $(0 \leq t \leq T-1)$ and $\varphi_0(s_i)$ to be the probability of starting in each state s_i. The recursion is given by:

$$\varphi_{t+1}(s_i) = \max_{s_j} \{\varphi_t(s_j)\exp\mathbf{F}(\mathbf{s}, \mathbf{o}, t+1)\} \tag{4}$$

The recursion stops when $t = T$ - 1 and the biggest unnormalized probability is $p_\theta^* = \text{argmax}_i[\varphi_T(s_i)]$. At this time, we can backtrack through the stored information to find the most likely sequence $\mathbf{s}^*$.

2.2 Training Conditional Random Fields

CRFs are trained by setting the set of weights $\theta = \{\lambda_1, \lambda_2, \ldots\}$ to maximize the log–likelihood, L, of a given training data set $\mathcal{D} = \{(\mathbf{o}^{(j)}, \mathbf{l}^{(j)})\}_{j=1}^{N}$:

$$L = \sum_{j=1}^{N} \log\Big(p_\theta(\mathbf{l}^{(j)}|\mathbf{o}^{(j)})\Big) = \sum_{j=1}^{N}\sum_{t=1}^{T} \mathbf{F}(\mathbf{l}^{(j)}, \mathbf{o}^{(j)}, t) - \sum_{j=1}^{N} \log Z(\mathbf{o}^{(j)}) \tag{5}$$

When the label sequences in the training dataset is complete, the likelihood function in exponential models such as CRFs is convex, thus searching the global optimum is guaranteed. However, the optimum can not be found analytically. Parameter estimation for CRFs requires an iterative procedure. It has been shown that quasi–Newton methods, such as L–BFGS [13], are most efficient [4]. This method can avoid the explicit estimation of the Hessian matrix of the log–likelihood by building up an approximation of it using successive evaluations of the gradient. L–BFGS is a limited–memory quasi–Newton procedure for unconstrained convex optimization that requires the value and gradient vector of the function to be optimized. The log–likelihood gradient component of λ_k is

$$\begin{aligned}\frac{\delta L}{\delta \lambda_k} &= \sum_{j=1}^{N}\left[\widetilde{C}_k(\mathbf{l}^{(j)}, \mathbf{o}^{(j)}) - \sum_{\mathbf{s}} p_\theta(\mathbf{s}|\mathbf{o}^{(j)}) C_k(\mathbf{s}, \mathbf{o}^{(j)})\right] \\ &= \sum_{j=1}^{N}\Big[\widetilde{C}_k(\mathbf{l}^{(j)}, \mathbf{o}^{(j)}) - \mathrm{E}_{p_\theta} C_k(\mathbf{s}, \mathbf{o}^{(j)})\Big]\end{aligned} \tag{6}$$

where $\widetilde{C}_k(\mathbf{l}^{(j)}, \mathbf{o}^{(j)}) = \sum_{t=1}^{T} f_k(\mathbf{l}_{t-1}^{(j)}, \mathbf{l}_t^{(j)})$ if λ_k is associated with an edge feature f_k and $= \sum_{t=1}^{T} g_k(\mathbf{o}^{(j)}, \mathbf{l}_t^{(j)})$ if λ_k is associated with a state feature g_k. Intuitively, it is the expectation (i.e., the count) of feature f_k (or g_k) with respect to the j^{th} training sequence of the empirical data $\mathcal{D}$. And $\mathrm{E}_{p_\theta} C_k(\mathbf{s}, \mathbf{o}^{(j)})$ is the expectation (i.e., the count) of feature f_k (or g_k) with respect to the CRF model p_θ.

The training process for CRFs requires to evaluate the log-likelihood function L and gradient vector $\{\frac{\delta L}{\delta \lambda_1}, \frac{\delta L}{\delta \lambda_2}, \ldots\}$ at each training iteration. This is very time-consuming because estimating the partition function $Z(o^{(j)})$ and the expected value $\mathrm{E}_{p_\theta} C_k(\mathbf{s}, \mathbf{o}^{(j)})$ needs an intensive forward-backward computation. This computation manipulates on the transition matrix M_t at every time position t of each data sequence. M_t is defined as follows,

$$M_t[l^{'}][l] = \exp\, \mathbf{F}(\mathbf{s}, \mathbf{o}, t) = \exp\,(\sum_i \lambda_i f_i(s_{t-1}, s_t) + \sum_j \lambda_j g_j(\mathbf{o}, s_t)) \tag{7}$$

To compute the partition function $Z(o^{(j)})$ and the expected value $\mathrm{E}_{p_\theta} C_k(\mathbf{s}, \mathbf{o}^{(j)})$, we need forward and backward vector variables α_t and β_t defined as follows,

$$\alpha_t = \begin{cases} \alpha_{t-1} M_t & 0 < t \leq T \\ \mathbf{1} & t = 0 \end{cases} \tag{8}$$

$$\beta_t^\top = \begin{cases} M_{t+1} \beta_{t+1}^\top & 1 \leq t < T \\ \mathbf{1} & t = T \end{cases} \tag{9}$$

$$Z(\mathbf{o}^{(j)}) = \alpha_T \mathbf{1}^\top \tag{10}$$

$$\mathrm{E}_{p_\theta} C_k(\mathbf{s}, \mathbf{o}^{(j)}) = \sum_{t=1}^{T} \frac{\alpha_{t-1}(f_k * M_t)\beta_t^\top}{Z(o^{(j)})} \tag{11}$$

3 Training CRFs on Multiprocessor Systems

3.1 The Need of Parallel Training of CRFs

In the sequential algorithm for training CRFs computing log-likelihood L and its gradient $\{\frac{\delta L}{\delta \lambda_1}, \frac{\delta L}{\delta \lambda_2}, \ldots\}$ is most time-consuming due to the heavy forward-backward computation on transition matrices. The L-BFGS update is very fast even if the log-likelihood function is very high dimensional. Therefore, the computational complexity of the training algorithm is mainly estimated from the former step.

The time complexity for calculating the transition matrix M_t in (7) is $\mathcal{O}(\bar{n}|\mathcal{L}|^2)$ where $|\mathcal{L}|$ is the number of class labels and $\bar{n}$ is the average number of *active* features at a time position in a data sequence. Thus, the time complexity to the partition function $Z(\mathbf{o}^{(j)})$ according to (8) and (10) is $\mathcal{O}(\bar{n}|\mathcal{L}|^2 T)$, in which T is the length of the observation sequence $\mathbf{o}^{(j)}$. And, the time complexity for computing the feature expectation $\mathrm{E}_{p_\theta} C_k(\mathbf{s}, \mathbf{o}^{(j)})$ is also $\mathcal{O}(\bar{n}|\mathcal{L}|^2 T)$. As a result, the time complexity for evaluating the log-likelihood function and its gradient vector is $\mathcal{O}(N\bar{n}|\mathcal{L}|^2 \bar{T})$, in which N is the number of training data sequences and T is now replaced by $\bar{T}$ - the average length of training data sequences. Because we train the CRF model m iterations, the final computational complexity of the serial training algorithm is $\mathcal{O}(mN\bar{n}|\mathcal{L}|^2 \bar{T})$. This computational complexity is for first-order Markov CRFs. If we use the second-order Markov CRFs in which the label of the current state depends on two labels of two previous states, the complexity is now proportional to $|\mathcal{L}|^4$, i.e., $\mathcal{O}(mN\bar{n}|\mathcal{L}|^4 \bar{T})$.

Although the training complexity of CRFs is polynomial with respect to all input parameters, the training process on large-scale datasets is still prohibitively expensive. In practical implementation, the computational time for training CRFs is even larger than what we can estimate from the theoretical complexity; this is because many other operations need to be performed during training, such as feature scanning, mapping between different data formats, numerical scaling (to avoid numerical problems), and smoothing. For example, training a first-order Markov CRF model for POS tagging ($|\mathcal{L}| = 45$) on about 1 million words (i.e., $N\bar{T} \simeq 1,000,000$) from the Wall Street Journal corpus (Penn TreeBank) took approximately 100 hours, i.e., more than 4 days.

In the point of view of machine learning, speeding up the training of CRFs is motivated by a couple of reasons. First, there are more large-scale annotated datasets in NLP and Bioinformatics. Further, unlike natural language sentences, biological data sequences are much longer, and thus need more time for training and inference. Second, generative models like CRFs can incorporate millions of

features. However, not all features are relevant. Feature selection for choosing most important and useful features from a huge set of candidates sometimes requires the model to be re-trained over and over again. Third, another challenge is that in many new application domains, the lack of labeled training data is very critical. Building large annotated datasets requires a lot of human resources. Semi-supervised learning is a way to build accurate prediction models using a small set of labeled data as well as a large set of unlabeled data because unlabeled data are widely available and easy to obtain. There are several approaches in semi-supervised learning like self- and co-training that also need the models to be trained again and again. Finally, building an accurate prediction model needs a repeated refinement because the learning performance of a model like CRF depends on different parameter settings. This means that we have to train the model several times using different values for input parameters and/or under different experimental setups till it reaches a desired output. Thus, accelerating the training process can save time for practitioners significantly.

3.2 The Parallel Training of CRFs

Input:
- Training data: $\mathcal{D} = \{(\mathbf{o}^{(j)}, \mathbf{l}^{(j)})\}_{j=1}^{N}$
- The number of parallel processes: P;
- The number of training iterations: m

Output:
- Optimal feature weights: $\theta^* = \{\lambda_1^*, \lambda_2^*, \ldots\}$

Initial Step:
- Generate features with initial weights $\theta = \{\lambda_1, \lambda_2, \ldots\}$
- Each process loads its own data partition $\mathcal{D}_i$

Parallel Training (each training iteration):
1. The root process broadcasts θ to all parallel processes
2. Each process P_i computes the local log-likelihood L_i and local gradient vector $\{\frac{\delta L}{\delta \lambda_1}, \frac{\delta L}{\delta \lambda_2}, \ldots\}_i$ on $\mathcal{D}_i$
3. The root process gathers and sums all L_i and $\{\frac{\delta L}{\delta \lambda_1}, \frac{\delta L}{\delta \lambda_2}, \ldots\}_i$ to obtain the global L and $\{\frac{\delta L}{\delta \lambda_1}, \frac{\delta L}{\delta \lambda_2}, \ldots\}$
4. The root process performs L-BFGS optimization search to update the new feature weights θ
5. If #iterations $< m$ then goto step 1, stop otherwise

Table 1. Parallel algorithm for training CRFs

As we can see from (5) and (6), the log-likelihood function and its gradient vector with respect to training dataset $\mathcal{D}$ are computed by summing over all training data sequences. This *nature sum* allows us to divide the training dataset into different partitions and evaluate the log-likelihood function and its gradient on each partition independently. As a result, the parallelization of the training process is quite straightforward.

How the Parallel Algorithm Works The parallel algorithm is shown in Table 1. The algorithm follows the master-slave strategy. In this algorithm, the training dataset $\mathcal{D}$ is randomly divided into P equal partitions: $\mathcal{D}_1, \ldots, \mathcal{D}_P$. At

the initialization step, each data partition is loaded into the internal memory of its corresponding process. Also, every process maintains the same vector of feature weights θ in its internal memory.

At the beginning of each training iteration, the vector of feature weights on each process will be updated by communicating with the master process. Then, the local log-likelihood L_i and gradient vector $\{\frac{\delta L}{\delta \lambda_1}, \frac{\delta L}{\delta \lambda_2}, \ldots\}_i$ are evaluated in parallel on distributed processes; the master process then gathers and sums those values to obtain the global log-likelihood L and gradient vector $\{\frac{\delta L}{\delta \lambda_1}, \frac{\delta L}{\delta \lambda_2}, \ldots\}$; the new setting of feature weights is updated on the master process using L-BFGS optimization. The algorithm will check for some terminating criteria to whether stop or perform the next iteration. The output of the training process is the optimal vector of feature weights $\theta^* = \{\lambda_1^*, \lambda_2^*, \ldots\}$. Although the stopping criteria can be difference in likelihood or in the norm of the parameter vector between two consecutive iterations, we usually use the iteration count because training CRFs according to those criteria might take too many iterations and of course suffer from overfitting problem.

Data Communication and Synchronization In each training iteration, the master process has to communicate with each slave process twice: (1) broadcasting the vector of feature weights and (2) gathering the local log-likelihood and gradient vector. These operations are performed using message passing mechanism. Let n be the number of feature weights and weights are encoded with "double" data type, the total amount of data needs to be transferred between the master and each slave is $8(2n+1)$. If, for example, $n = 1,500,000$, the amount of data is approximately 23Mb. This is very small in comparison with high-speed links among computing nodes on massively parallel processing systems. A barrier synchronization is needed at each training iteration to wait for all processes complete their estimation of local log-likelihood and gradient vector.

Data Partitioning and Load Balancing Load balancing is important to parallel programs for performance reasons. Because all tasks are subject to a barrier synchronization point at each training iteration, the slowest process will determine the overall performance. In order to keep a good load balance among processes, i.e., to reduce the total idle time of computing processes as much as possible, we attempt to divide data into partitions as equally as possible. Let $M = \sum_{j=1}^{N} |\mathbf{o}^{(j)}|$ be the total number of data observations in training dataset $\mathcal{D}$. Ideally, each data partition $\mathcal{D}_i$ consists of N_i data sequences having exactly $\frac{M}{P}$ data observations. However, this ideal partitioning is not always easy to find because the lengths of data sequences are different. To simplify the partitioning step, we accept an approximate solution as follows. Let δ be some integer number, we attempt to find a partitioning in which the number of data observations in each data partition belongs to the interval $[\frac{M}{P} - \delta, \frac{M}{P} + \delta]$. To search for the first acceptable solution, we follow the round-robin partitioning policy in which longer data sequences are considered first. δ starts from some small value and will be gradually increased until the first solution is satisfied.

4 Empirical Evaluation

We performed two important natural language processing tasks, text noun phrase chunking and all-phrase chunking, on large-scale datasets to demonstrate two main points: (1) the large reduction in computational time of the parallel training of CRFs on massively parallel computers in comparison with the serial training; (2) when being trained on large-scale datasets, CRFs tends to achieve higher prediction accuracy in comparison with the previous applied learning methods.

4.1 Experimental Environment

The experiments were carried out using our C/C++ implementation[3] of second-order Markov CRFs. It was designed to deal with hundreds of thousand data sequences and millions of features. It can be compiled and run on any parallel system supporting message passing interface (MPI). We used a Cray XT3 system (Linux OS, 180 AMD Opteron 2.4GHz processors, 8GB RAM per each, high-speed (7.6GB/s) interconnection among processors) for the experiments.

4.2 Text Chunking

Text chunking[4], an intermediate step towards full parsing of natural language, recognizes phrase types (e.g., noun phrase, verb phrase, etc.) in input text sentences. Here is a sample sentence with phrase marking: "[NP *Rolls-Royce Motor Cars Inc.*] [VP *said*] [NP *it*] [VP *expects*] [NP *its U.S. sales*] [VP *to remain*] [ADJP *steady*] [PP *at*] [NP *about 1,200 cars*] [PP *in*] [NP *1990*]."

4.3 Text Chunking Data and Evaluation Metric

We evaluated NP chunking and chunking on two datasets: (**1**) **CoNLL2000-L**: the training dataset consists of 39,832 sentences of sections from 02 to 21 of the Wall Street Journal (WSJ) corpus and the testing set includes 1,921 sentences of section 00 of WSJ; and (**2**) **25-fold CV Test**: 25-fold cross-validation test on all 25 sections of WSJ. For each fold, we took one section of WSJ as the testing set and all the others as training set.

Label representation for phrases is either IOB2 or IOE2. B indicates the beginning of a phrase, I is the inside of a phrase, E marks the end of a phrase, and O is outside of all phrases. The label path in IOB2 of the sample sentence is "B-NP I-NP I-NP I-NP B-VP B-NP B-VP B-NP I-NP I-NP B-VP I-VP B-ADJP B-PP B-NP I-NP I-NP B-PP B-NP O".

Evaluation metrics are precision ($pre. = \frac{a}{b}$), recall ($rec. = \frac{a}{c}$), and $F_{\beta=1} = 2 \times (pre. \times rec.)/(pre. + rec.)$; in which a is the number of *correctly* recognized phrases (by model), b is is the number of recognized phrases (by model), and c is the the number of actual phrases (by humans).

[3] PCRFs: http://www.jaist.ac.jp/∼hieuxuan/flexcrfs/flexcrfs.html
[4] See the CoNLL-2000 shared task: http://www.cnts.ua.ac.be/conll2000/chunking

4.4 Feature Selection for Text Chunking

To achieve high prediction accuracy on these tasks, we train CRF model using the second-order Markov dependency. This means that the label of the current state depends on the labels of the two previous states. As a result, we have four feature types as follows rather than only two types in first-order Markov CRFs.

$$f_i(s_{t-1}, s_t) \equiv [s_{t-1} = l^{'}][s_t = l]$$
$$g_j(\mathbf{o}, s_t) \equiv [x_i(\mathbf{o}, t)][s_t = l]$$
$$f_k(s_{t-2}, s_{t-1}, s_t) \equiv [s_{t-2} = l^{''}][s_{t-1} = l^{'}][s_t = l]$$
$$g_h(\mathbf{o}, s_{t-1}, s_t) \equiv [x_h(\mathbf{o}, t)][s_{t-1} = l^{'}][s_t = l]$$

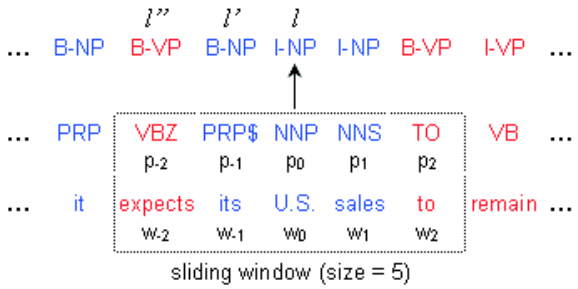


Fig. 1. An example of a data sequence

$w_{-2},\ w^*_{-1},\ w^*_0,\ w_1,\ w_2,\ w_{-1}w^*_0,\ w_0w_1$
$p_{-2},\ p^*_{-1},\ p^*_0,\ p_1,\ p_2,\ p_{-2}p_{-1},\ p_{-1}p^*_0,\ p_0p_1,\ p_1p_2$
$p_{-2}p_{-1}p_0,\ p_{-1}p_0p_1,\ p_0p_1p_2,\ p_{-1}w^*_{-1},\ p_0w^*_0$
$p_{-1}p_0w^*_{-1},\ p_{-1}p_0w^*_0,\ p_{-1}w_{-1}w^*_0,\ p_0w_{-1}w^*_0,\ p_{-1}p_0p_1w_0$

Table 2. Context predicate templates for text chunking

Figure 1 shows a sample training data sequence for text chunking. The top half is the label sequence and the bottom half is the observation sequence including tokens (words or punctuation marks) and their POS tags. Table 2 describes the context predicate templates for text chunking. Here w denotes a token; p denotes a POS tag. A predicate template can be a single token (e.g., the current word: w_0), a single POS tag (e.g., the POS tag of the previous word: p_{-1}), or a combination of them (e.g., the combination of the POS tag of the previous word, the POS tag of the current word, and the current word: $p_{-1}p_0w_0$). Context predicate templates with asterisk ($*$) are used for both state feature type 1 (i.e., g_j) and state feature type 2 (i.e., g_h). We also apply rare (cut-off) thresholds for both context predicates and state features (the threshold for edge features is zero). Those predicates and features whose occurrence frequency is smaller than 2 will be removed from our models to reduce overfitting.

4.5 Experimental Results of Text Chunking

Methods	NP chunking $F_{\beta=1}$	All-phrase chunking $F_{\beta=1}$
Ours (majority voting among 16 CRFs)	**96.74**	**96.33**
Ours (CRFs, about 1.3M - 1.5M features)	**96.59**	**96.18**
Kudo & Matsumoto 2001 (voting SVMs)	95.77	–
Kudo & Matsumoto 2001 (SVMs)	95.34	–
Sang 2000 (system combination)	94.90	–

Table 3. Accuracy comparison of NP and all-phrase chunking on the CoNLL2000-L

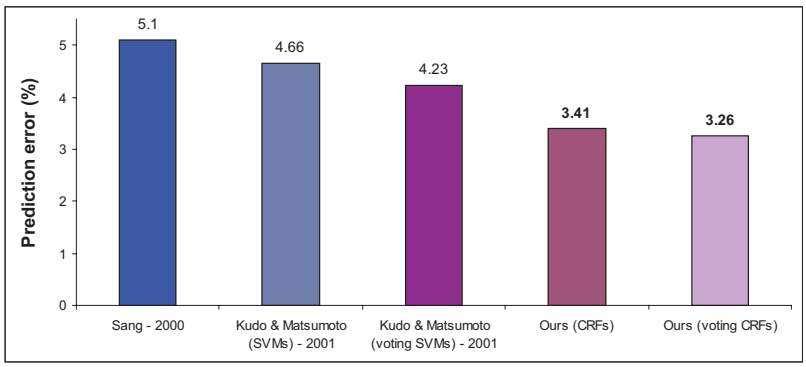


Fig. 2. Error rate comparison for noun phrase chunking on CoNLL2000-L dataset

Datasets	n_{00}	n_{01}	n_{10}	n_{11}	χ^2	χ	Null hypothesis
CoNLL2000-L	304	336	525	45286	41.0499	6.4070	REJECT

Table 4. Statistical (McNemar) test for comparing prediction models: between our second-order Markov **CRFs** and **SVMs** (Kudo & Matsumoto 2001)

Table 3 shows the comparison of F_1-scores of NP and all-phrase chunking tasks on the CoNLL2000-L dataset among state-of-the-art chunking systems. Figure 2 shows our improvement in accuracy in comparison with the previous work in a more visual way. Our model reduces error by 22.93% on NP chunking relative to the previous best system, i.e., Kudo & Matsumoto's work.

We performed McNemar's test for noun phrase chunking on the CoNLL2000-L dataset mentioned above. The statistical test was done between our second-order Markov CRFs and SVMs (Kudo & Matsumoto 2001) [17]. Table 4 shows

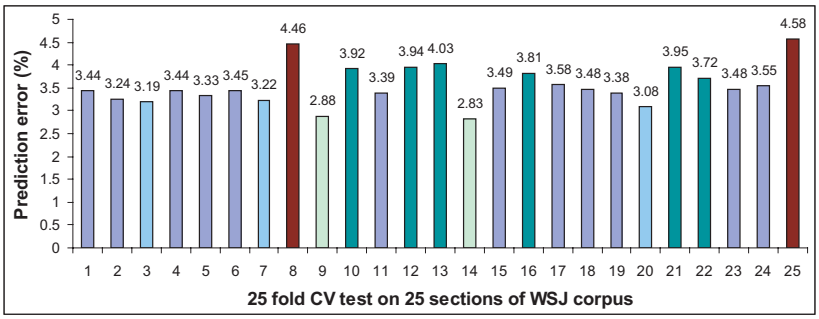


Fig. 3. 25-fold cross-validation test of NP chunking on the whole 25 sections of WSJ

all information about the McNemar's tests: n_{00} be the number of examples (i.e., words) in the testing set that are misclassified by both our CRF and SVM models; n_{01} be the number of examples misclassified by our CRF but not by SVM; n_{10} be the number of examples misclassified by SVM but not by our CRF; and n_{11} be the number of misclassified by neither our CRF nor SVM. Where $n = n_{00} + n_{01} + n_{10} + n_{11}$ is the total number of examples in the testing set. This test takes into account the number of examples misclassified by our CRF model when compared to the SVM model of Kudo & Matsumoto and vice-versa. The null hypothesis (i.e., H_0) is that both prediction models have equal error rate, i.e., $n_{01} = n_{10}$. We have the chi-square statistic with one degree of freedom,

$$\chi^2 = \frac{(|n_{01} - n_{10}| - 1)^2}{n_{01} + n_{10}}$$

McNemar's test accepts the null hypothesis at significance level α if $\chi^2 \leq \chi^2_{\alpha,1}$. The typical value for α is 0.05 and $\chi^2_{0.05,1} = 3.84$. Since the null hypothesis is rejected and n_{01} is greater than n_{10}, we conclude that our CRF models outperform the SVM models of Kudo & Matsumoto on this task.

In order to investigate chunking performance on the whole WSJ, we performed a 25-fold CV test on all 25 sections. We trained totally 50 CRF models for 25 folds for NP chunking using two label styles IOB2, IOE2 and only one initial value of θ (= .00). The number of features of these models are approximately 1.5 million. Figure 3 shows the lowest error rates of those 25 sections.

4.6 Computational Time Measure and Analysis

We also measured the computational time of the CRF models the Cray XT3 system. For example, training 130 iterations of NP chunking task on CoNLL2000-L dataset using a single process took 38h57' while it took only 56' on 45 parallel processes. Similarly, each fold of the 25-fold CV test of NP chunking took an average training time of 1h21' on 45 processes while it took approximately 56h on one process. All-phrase chunking is much more time-consuming. This is because the number of class labels is $|\mathcal{L}| = 23$ on CoNLL2000-L. For example, serial training on the CoNLL2000-L requires about 1348h for 200 iterations (i.e., about

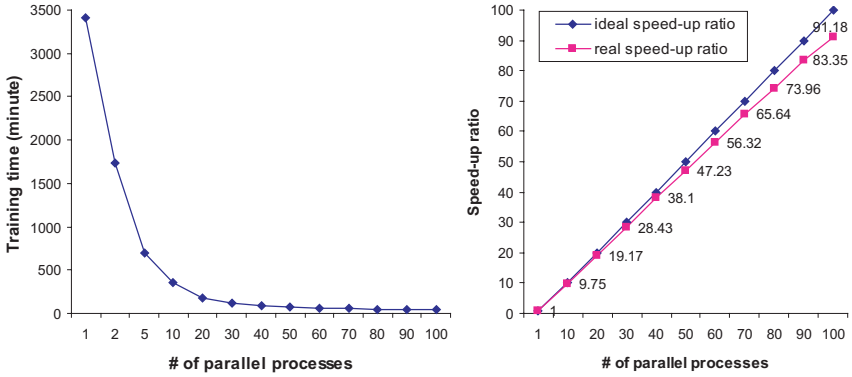


Fig. 4. The computational time of parallel training and the speed-up ratio of the first fold (using IOB2) of 25-fold CV test on WSJ

56 days) whereas it took only 17h46' on 90 parallel processes. Figure 4 depicts the computational time and the speed-up ratio of the parallel training CRFs on the Cray XT3 system.

5 Conclusions

We have presented a high-performance training of CRFs on large-scale datasets using massively parallel computers. And the empirical evaluation on text chunking with different data sizes and parameter configurations shows that second-order Markov CRFs can achieved a significantly higher accuracy in comparison with the previous results, particularly when being provided enough computing power and training data. And, the parallel training algorithm for CRFs helps reduce computational time dramatically, allowing us to deal with large-scale problems not limited to natural language processing.

References

1. Pinto, D. McCallum, A, Wei, X., and Croft, B. (2003). Table extraction using conditional random fields. The 26th ACM SIGIR.
2. Kristjansson, T., Culotta, A., Viola, P., and McCallum, A. (2004). Interactive information extraction with constrained conditional random fields. The 19th AAAI.
3. Cohn, T., Smith, A., and Osborne, M. (2005). Scaling conditional random fields using error-correcting codes. The 43th ACL.
4. Sha, F. and Pereira, F. (2003). Shallow parsing with conditional random fields. HLT/NAACL.
5. Kumar, S. and Hebert, M. (2003). Discriminative random fields: a discriminative framework for contextual interaction in classification. The IEEE CVPR.
6. Quattoni, A., Collins, M., and Darrell, T. (2004) "Conditional random fields for object recognition", The 18th NIPS.

7. Torralba, A., Murphy, K., and Freeman, F. (2004) "Contextual models for object detection using boosted random fields", The 18th NIPS.
8. He, X., Zemel, R.S., and Carreira-Perpinan, M.A. (2004) "Multiscale conditional random fields for image labeling", The IEEE CVPR.
9. Lafferty, J., Zhu, X., and Liu, Y. (2004) "Kernel conditional random fields: representation and clique selection", The 20th ICML.
10. Liu, Y., Carbonell, J., Weigele, P., and Gopalakrishnan, V. (2005) "Segmentation conditional random fields (SCRFs): a new approach for protein fold recognition", The 9th RECOMB.
11. Lafferty, J., McCallum, A., and Pereira, F. (2001) "Conditional random fields: probabilistic models for segmenting and labeling sequence data", The 18th ICML.
12. Rabiner, L. (1989) "A tutorial on hidden markov models and selected applications in speech recognition", Proc. of IEEE, vol.77, no.2, pp. 257-286.
13. Liu, D. and Nocedal, J. (1989) "On the limited memory bfgs method for large-scale optimization", Mathematical Programming, vol.45, pp. 503-528.
14. Pietra, S.D., Pietra, V.D., and Lafferty, J. (1997) "Inducing features of random fields", IEEE PAMI, 19(4):380–393.
15. McCallum, A. (2003) "Efficiently inducing features of conditional random fields", The 19th UAI.
16. Sang, E. (2000) "Noun phrase representation by system combination", The ANLP/NAACL.
17. Kudo, T. and Matsumoto, Y. (2001) "Chunking with support vector machines", The NAACL.
18. Chen, S. and Rosenfeld, R. (1999) "A gaussian prior for smoothing maximum entropy models", Technical Report CS-99-108, CMU.

A Parallel Feature Selection Algorithm from Random Subsets

Daniel J. Garcia, Lawrence O. Hall, Dmitry B. Goldgof, and Kurt Kramer

Department of Computer Science and Engineering
4202 E. Fowler Ave. ENB118
University of South Florida
Tampa, FL 33620, USA
(djgarcia,hall,goldgof,kkramer)@csee.usf.edu

Abstract. Feature selection methods are used to find the set of features that yield the best classification accuracy for a given data set. This results in lower training and classification time for a classifier, a support vector machine here, and better classification accuracy. Feature selection, however, may be a time consuming process unfit for real time application. In this paper, we explore a feature selection algorithm based on support vector machine training time. It is compared with the Wrapper algorithm. Our approach can be run on all available processors in parallel. Our feature selection approach is ideal if new features need to be selected during data acquisition, where a fast, approximate approach may be advantageous. Experimental results indicate that the training time based method yields feature sets almost as good as the Wrapper method, while requiring considerably less computation time.

Keywords: Feature Selection, Parallelism, Random Subsets, Wrappers, SVM.

1 Introduction

Support vector machines (SVMs)[1] are learning algorithms which result in a model that can be used to classify data. The details of the inner workings of SVMs are beyond the scope of this paper. For our purposes, SVMs use labeled data to construct a classifier, and then use it to classify unknown data. In this paper, the data being analyzed are plankton images obtained from a device called SIPPER (Shadow Image Particle Profiling Evaluation Recorder) [2]. In order for a support vector machine to be able to classify these images, features are extracted from them. These features are used by the SVM to create the support points that will differentiate between images. These features can be numerous and include characteristics such as height, weight, shape, length, transparency, and texture.

The use of all the available features does not guarantee the best accuracy, training, and classification time. It is possible for a subset of features to have better accuracy, training, and classification time. Also of importance is the fact

that the process of training a SVM is faster if fewer features are used. It is for this reason that feature selection processes are necessary in order to effectively train a classifier. Feature selection is the process through which an "optimal" group of features for a given data set is found. This process may be a time consuming one, and is typically not well suited for real time applications. The goal is to create a feature selection algorithm that is able to complete its execution in a short enough amount of time to allow it to be implemented in the field during data acquisition, while retaining high classification accuracy. Our hypothesis is that sets of features which result in faster SVM training times can be exploited to create overall feature sets which can be used to build a high accuracy classifier. It is expected that less training time will be required to find the boundaries with features that will enable higher accuracy classifiers to be built.

There are many feature selection methods. In fact, there are a number which are relatively specific for support vector machines [3, 4]. The recursive feature elimination approach has been used with success with SVM's [4]. Space limitations prevent us from doing detailed comparisons, but we do present an alternative, feature selection approach that works with SVM's.

2 Random Feature Selection

Feature selection methods are applied to all the features describing a data set to find a subset of features that best describe that data set. The feature selection method proposed in this paper can be divided into two stages. The first stage consists of generating a number of feature sets of fixed size, then running a 10 fold cross validation using only the features found in these sets, to determine how well they are able to classify the data. It is important to emphasize that the features in these sets are randomly selected out of the pool of all features, and thus these sets are generated in a very short amount of time. The sets of features are then sorted by a given criteria, such as training time (here) or the number of support vectors generated, and the best of these randomly generated sets are selected for the second stage of the algorithm.

For the second stage of the method we have a number of ranked feature sets. Using these sets, a new set composed of the union of the features found in the selected sets is created. At this point, the classifier is trained using the newly created feature sets, and then it is tested against a previously unseen test set to see how well it performs. The number of feature sets selected for the second stage of the method can vary from 2 to the number of sets generated during the first stage of the process. Figure 1 shows a flowchart of the random sets feature selection method. The algorithm had minimal sensitivity to increasing the number of feature sets. The choice of numbers of feature sets to union needs more exploration with the goal of smaller numbers in the union resulting in fewer chosen features.

One very distinct characteristic of the random sets approach is that the random sets are all independent of each other. Feature selection algorithms usually go through a large number of possible combinations of features in order to find

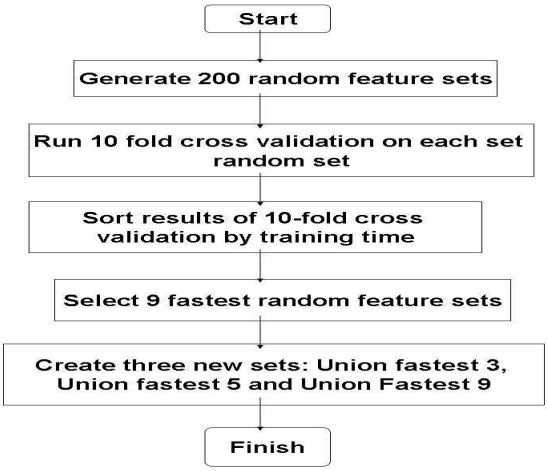


Fig. 1. Random Sets Flow Chart

the best one. However, since the number of features could possibly reach the hundreds, the number of possible combinations grows at a very fast rate. It is for this reason that existing feature selection algorithms do not search for possible combinations blindly, instead they do it intelligently. This means that there is some logic guiding the search, and the next step in the search process is partly based on previous steps. The implications of this characteristic is that these feature selection methods cannot fully take advantage of parallel processing because future steps in the process need to wait for previous steps to finish. The random sets method, on the other hand, does allow parallel processing. The random feature sets created are completely independent from each other, and all of them are evaluated during the same step in the algorithm. For this reason, it is possible for every single random set created to be evaluated in parallel; greatly reducing the time it takes for this feature selection method to finish its task. In this work, timings are reported with all training done on 1 processor. One could divide by approximately the number of random feature sets evaluated (there will be some overhead) to look at the parallel computing time advantage. The reader will see the speed-ups are quite impressive even without parallelism.

3 Wrappers

An alternative algorithm for feature selection is also presented. Feature selection methods usually work by trying combinations of features from the original pool of features and then choosing the combination that yields the best results. One such method consists of organizing the feature combinations in a tree structure. In this organization, the nodes of the tree are simply a given combination of features. This is the approach taken by the Wrapper feature selection method [5].

A given combination of features is passed to a learning algorithm for evaluation and then the results are obtained and kept for later comparison. The results from the learning algorithm are then used to intelligently search the tree structure.

To illustrate this approach, let us assume we have n features. The root of the tree is the set containing all n features. This set is analyzed first and the accuracy is stored. The next step in the algorithm is to choose the best stored results and then analyze every combination of the number of features in that set minus one. In the current case, only the results of analyzing one set have been obtained, so the next step would be to analyze every combination of n -1 features and store the results. At this point, a best first search is done, which means that the best case, gauged by the classification accuracy, is selected in order to repeat the process of searching for every combination of features of length s - 1, where s is the number of features in the most recently selected feature set. Thus, the next step is to look at all (n - 1) - 1 = n - 2 subsets of the best n - 1 grouping.

The stopping criterion was the analysis of a fixed number of new feature sets without finding a new set with clearly higher accuracy [6]. We did allow sub-optimal (5^{th} best) sets to be examined with a low probability.

This feature selection method can't take full advantage of parallel processing. This is so because the search has to wait for the results of all the processed combinations before it can select the best next case. Suppose combinations of size s are being analyzed, all of these combinations come from a parent of size s + 1. Theoretically, every combination of size s in this case can be processed at the same time if enough processors are present. However, the result from these sets will be considered for the next best case, which forces this method to wait until every set of size s is evaluated before it can continue.

4 Data Set and Parameters

The results presented in this paper were obtained from experiments using a data set made up of plankton images obtained from the SIPPER device. The data set includes 5 different classes and consists of 8440 images total, with 1688 images per class. This image set was split into three smaller subsets for the purpose of the experiments. Two of the subsets contain 1000 images, with 200 images per class; and the remaining one contains 6440 images, with 1288 images per class. Feature selection is done on one of the sets with 1000 images; the Test Set is the other set with 1000 images; and the Training Subset had 6440 images. The data is these files has been stratified for 10 fold cross validations. The SVM used as the classifier is a modified version of libSMV [7], the parameters for the SVM are C = 16, Gamma = 0.04096, and the Gaussian radial basis function (RBF) was used as the kernel. The sequential minimal optimization (SMO) algorithm was the optimization algorithm used. For more information regarding the parameter tuning process and the RBF kernel, please refer to [8].

5 Experiments and Results

The procedure for the experiments was the following. First, from the original pool of 47 features extracted from the plankton images, 200 random subsets with 10 features each were created. We did some empirical tests and found these numbers to be the best of a range of reasonably equivalent choices. Clearly, the selection will make a difference.

A ten fold cross validation was done on the Feature Selection data set using each one of these sets independently, and the time it took to train the SVM using these sets was recorded. Next, the 9 feature sets associated with the shortest training times were selected for the second stage of our method. Three new sets were created by using the union of the features found in the selected sets: the union of the 3 sets, the union of the 5 sets, and the union of the 9 sets, respectively associated with the ordered shortest training time. Finally, the Feature Selection data set and the Training Subset data set are put together into a joint data set and a classifier is trained on this joint set using the three new feature sets. Then the classifier is tested against the unseen Test Set to obtain the final results.

The whole procedure is repeated five times with five different randomly chosen stratified sets of data. The results reported are the average values of the five experiments.

For the Wrapper approach, the procedure was the following. First, a search was performed on the Feature Selection set using a 5 fold cross validation to evaluate the feature sets. This will yield an accuracy value for each level in the search tree, thus we get an accuracy value for the best sets of n features, where n goes from 1 feature to all features. Next, the union of the Training Subset and the Feature Selection set are used to train the classifier using the best set of features at a specific level in the tree, and then the classifier is tested against the Test Set to determine how accurate it is on unseen data.

As with the random sets approach, this whole procedure is repeated five times over the same data sets as the random sets method and the results reported are the average values of the five different experiments.

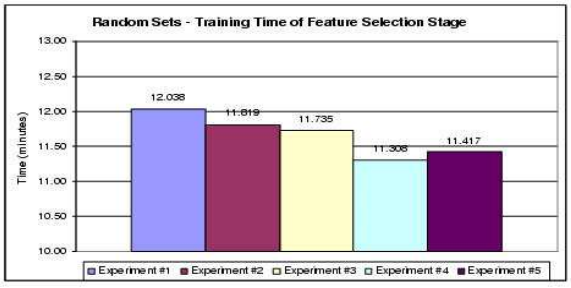


Fig. 2. Random Sets Average Training Time of Feature Selection Stage

The most important aspect of the random sets method is how fast it is. Figures 2 and 3 are graphs of the average training time of the feature selection stage for both the random sets approach, and the Wrappers approach for the five experiments. The training time of the feature selection stage of the random sets approach consists of the time it takes to train on the n random feature sets of fixed size, in this particular case, 200. The training time of the feature selection stage of the Wrapper method consists of the time it takes to train all of the combinations of features the Wrapper method tries while looking for an "optimal" set of features.

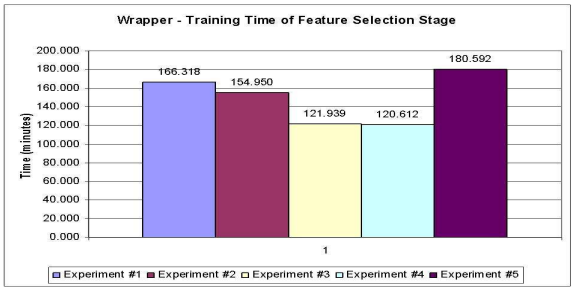


Fig. 3. Wrapper Average Training Time of Feature Selection Stage

There is a significant time difference between these two algorithms. The difference in time may be attributed to several factors, but chief among them, is the amount of work that each algorithm must do before finding their best feature sets. A good indicator of the amount of work each algorithm performs is the number of feature combinations evaluated during the search. The Wrapper approach consists of intelligently searching combinations of features starting with all features and reducing the number of features in the combinations as it progresses. The average number of combinations evaluated by the 5 iterations of the Wrapper method done here was 9372. Meanwhile, by the very definition of the random sets method, a fixed number of random feature sets needs to be evaluated. For the experiments carried out in this paper, only 200 random combinations were attempted for each of the five iterations, thus the average number of combinations across the five repetitions of the experiment is 200.

Specifically, the random sets approach shows a distinctly shorter training time during feature selection, and moderately shorter evaluation time when compared to the Wrapper. The average training time of the 200 random sets across the 5 individual experiments is approximately 700 seconds, or 11 minutes and 40 seconds. On the other hand, the average training time of all the combinations tried by the Wrapper approach across the 5 iterations of the experiment is approximately 9000 seconds, or 150 minutes. Thus, on average, there is a difference of approximately 2 hours and 19 minutes in time between the two feature selection methods.

The difference in evaluation time is not as great as the difference between the training times. Across the 5 experiments, the average evaluation time for the 200 randomly created sets is 43 seconds. The Wrapper, on the other hand, spent on average 2498 seconds, or 41 minutes and 38 seconds, on evaluation time. Adding the training time and the evaluation time together we get the total CPU time spent on each method. The random sets method spent an average of 744 seconds, or roughly 12 minutes to complete; on the other hand, the Wrapper method spent an average of 11,431 seconds, roughly 191 minutes, or 3 hours and 11 minutes, to finish.

Figure 4 shows a graph of the average accuracy for the union of the 3 fastest sets found by the random sets method. Because the set resulting from the union of these three sets contained 23 features, the accuracy is being compared with the average accuracy obtained by the Wrapper approach at 23 features. The remaining accuracy in Figure 4 is the best observed accuracy across all experiments, that is, taking into consideration both the random sets approach and the Wrapper approach. This accuracy was achieved by the Wrapper approach using the best 42 features found during the search and it is provided as a measure of how close the individual methods get to the best possible accuracy. Figure 5 provides training time measurements for these sets.

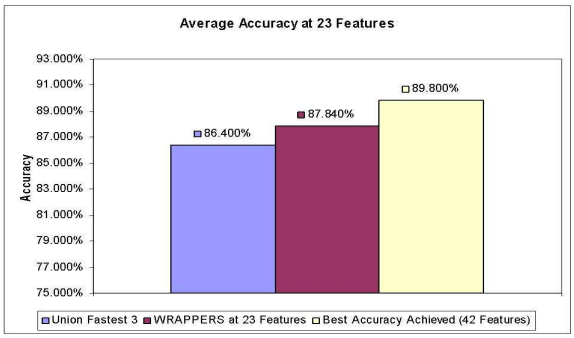


Fig. 4. Average Accuracy at 23 Features

At 23 features, the Wrapper approach was more accurate than the random sets approach by 1.44%. In turn, the best accuracy achieved was superior to the union of the fastest 3 random sets by 3.4%, and superior to Wrappers at 23 features by 1.96%. Figure 5 shows the average training times for the sets shown in Figure 4. Not surprisingly, the 23 features found by the Wrapper method are, on average, faster for training than both the 23 features found by the random sets, due to the Wrappers' ability to likely find the best 23 features, and the set of all features. The feature set found by the random sets is also faster for training than the set of all the features.

Figure 6 is a graph of the average accuracy values for the experiments involving the union of the 5 random feature sets that resulted in the fastest training

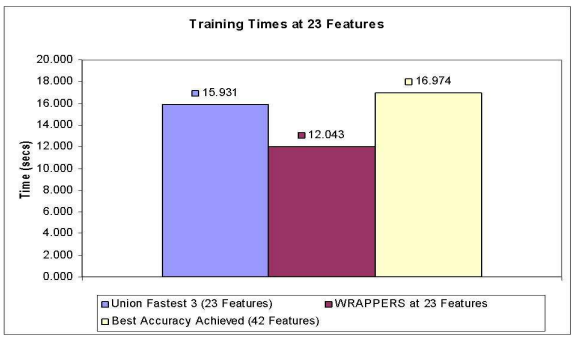


Fig. 5. Training Time at 23 Features

times. Using the fastest 5 random feature sets to train , the number of features in the union of these sets has increased to 31. For this reason, the accuracy of the union of the 5 fastest sets is being compared to the average accuracy of the Wrapper at 31 features. The best achieved accuracy is also provided for comparison purposes. Figure 7 shows the training time information for these sets.

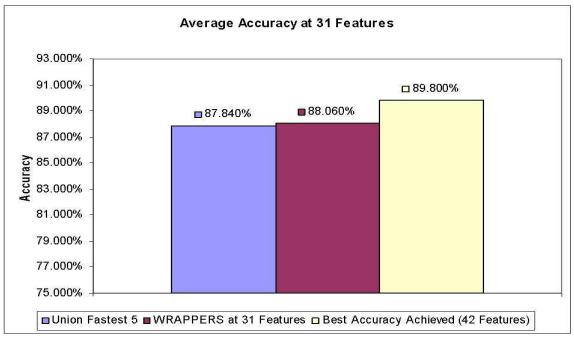


Fig. 6. Average Accuracy at 31 Features

As Figure 6 shows, the union of the 5 fastest to train sets, using 31 features, is inferior to Wrappers at the same number of features by only 0.22%. At this point, the best accuracy achieved is only 1.96% more than the random sets method, and 1.74% above the Wrappers. Figure 7 shows the training times for these three sets. Once again, the features found by the Wrappers were faster than the features found by the random sets.

Figure 8 is a graph of the final step of the random sets approach, when all the selected feature sets, in this case 9, were taken together to form a set consisting of the union of all the features in these sets. This new set contains 40 features; its

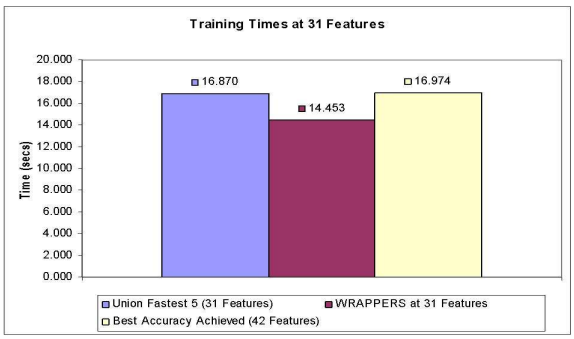


Fig. 7. Training Time at 31 Features

average accuracy is being compared with the average accuracy of the Wrappers at 42 features, and with the best accuracy achieved. Figure 9 is a graph of the average training times of the relevant feature sets.

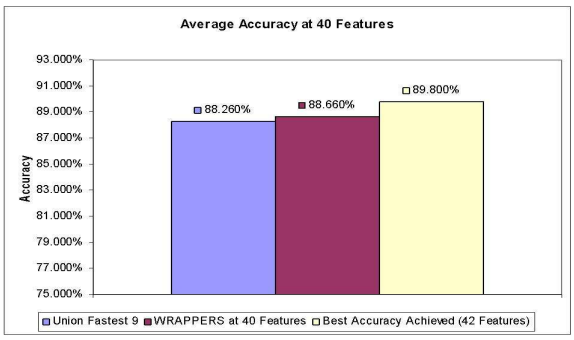


Fig. 8. Average Accuracy at 42 Features

As can be seen in Figure 8, with 40 features, the Wrapper method is only 1.14% less accurate than the best achieved accuracy. The random sets method, using the 40 features it found, is only 1.54% less accurate than the best accuracy obtained. Figure 9 shows a more interesting result, with 2 fewer features, the union of the fastest 9 sets has a higher training time than Wrappers at 42 features, that is, the best set of features found throughout the experiments. The reason for this behavior, as was stated previously, is that some features, in actuality, hinder the training process by making less clear the boundary between classes of images.

Figures 4, 6 and 8 show an interesting trend where the Wrapper approach performs slightly better than the random sets approach, and the sets of features that the Wrapper method produces are slightly faster for the training process

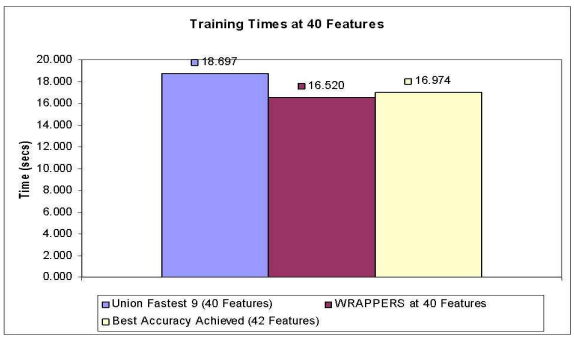


Fig. 9. Training Time at 42 Features

also. The reasoning is that the Wrapper method is a deeper, logically driven search, while our approach has a random element. This means that for any particular n, the Wrapper method should have approximated the best set consisting of n features, while the random sets could have found those features, it is not highly likely that it did. The advantage of the random sets method is that it finds sets of features that almost mirror the performance of the sets found by the Wrapper, but it does so in **considerably less time**. Time saving is the greatest asset of the random set method.

The random sets method is based on the hypothesis that the features that allow a SVM to train faster on a specific set of data are, in fact, the features better suited for that particular set of data. To test this hypothesis, the inverse of the hypothesis was used as the basis of the random set method and applied to one of the five data sets created for the random sets experiments. Using the inverse of the hypothesis implies selecting the "best" feature sets based on the fact they take the longest time to train; thus instead of selecting the fastest, to train, 9 sets to take to the second stage of the random sets method, the slowest 9 sets were selected. The results obtained from this experiment are compared to the result obtained from using the previously described random sets method on the same data set.

Figure 10 shows the accuracy for the union of 3 sets, the union of 5 sets, and the union of 9 sets as we use the fastest 9 random sets and the slowest 9 random sets. As can be seen in Figure 10, when the union operation is performed on the fastest sets, the accuracy is significantly higher in all three cases. The superiority of the features is shown in Figure 11, which gives the number of features in the relevant sets. Notice how the union of the fastest, to train, 3 sets actually has 3 fewer features than the union of the slowest 3 sets, however it is 10.6% more accurate. The accuracy continues to be higher for the union of the fastest 5 sets, and the union of the fastest 9 sets, however, the difference in accuracy becomes smaller as the number of features involved increases.

Figure 12 is a graph of the average accuracies of the random sets method and the Wrapper Method vs. the number of features in each of the sets. The accuracy

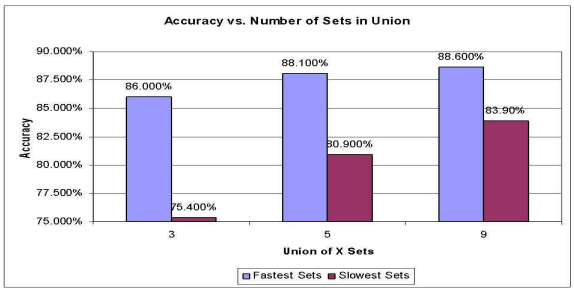


Fig. 10. Accuracy of Union vs. Number of Sets in Union

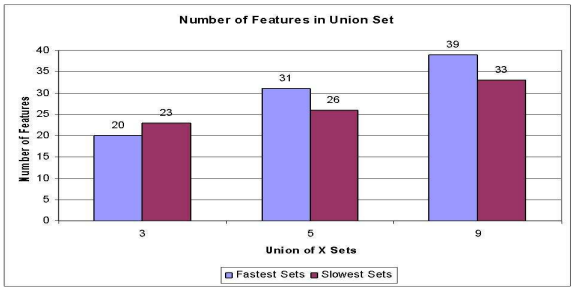


Fig. 11. Number of features in set vs. Number of Sets in Union

curve for the Wrapper method shows an increase as the number of features increases, reaching the highest average accuracy at 40 features. The random sets method is represented by three points, each representing the average accuracy of the union of the fastest 3, fastest 5, and fastest 9 random feature sets. Figure 12 clearly shows that the random sets method is able to find feature sets which can be used to create classifiers of comparable accuracy to those found by the Wrapper method, with the advantage that it does so in much less time.

6 Conclusion

As has been shown, using random feature sets as a feature selection tool provides benefits for learning algorithms. Real time application is one of the greatest benefits, perhaps allowing a limited feature selection algorithm to be run as new data is gathered. The random set approach is fast, can result in a very accurate classifier, and takes great advantage of available parallel processing. Each feature set can be evaluated in parallel. The Wrapper approach, on the other hand, was much slower but consistently more accurate. If accuracy is of the utmost importance, and feature selection time is no issue, the Wrapper method should be used; however, if time is critical, the random sets approach provides competitive accuracy while taking much less time.

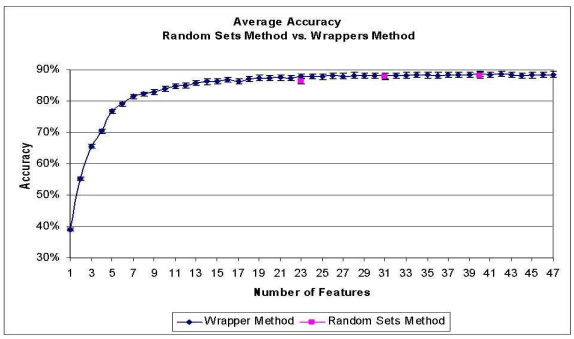


Fig. 12. Average Accuracy Random Sets Method vs. Wrapper Method

A real time application of the random set approach is the analysis of plankton on a cruise. The random set approach allows fast feature selection as different organisms are encountered. It is true that the accuracy will likely be slightly less than the best possible, but the difference in accuracy does not appear to be significant and it does allow for near real time optimization.

Acknowledgements: This research was partially supported by the United States Navy, Office of Naval Research, under grant number N00014-02-1-0266, the NSF under grant EIA-0130768 and by the Department of Energy through the Advanced Strategic Computing Initiative (ASCI) Visual Interactive Environment for Weapons Simulation (VIEWS) Data Discovery Program Contract number: DEAC04-76DO00789.

References

1. Tong Luo, Kurt Kramer, Dmitry B. Goldgof, Scott Samson Lawrence O. Hall, Andrew Remsen, and Thomas Hopkins. Recognizing Plankton from Shadow Image Particle Evaluation Recorder. *IEEE trans. on system, man and cybernetics-part B:cybernetics*, 34(4), 2004.
2. S. Samson, T. Hopkins, A. Remsen, L. Langebrake, T. Sutton, and J. Patten. A System for High Resolution Zooplankton Imaging. *IEEE Journal of Oceanic Engineering*, 26(4):671–676, 2001.
3. I. Guyon, S. Gunn, M. Nikravesh, and L.A. Zadeh, editors. *Feature Extraction Foundations and Applications*. Springer, 2006.
4. I. Guyon, J. Weston, and S. Barnhill. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46(1-3):389–422, 2002.
5. Ron Kohavi and George H. John. Wrappers for Feature Subset Selection. *Artificial Intelligence Archive*, 97:273–324, 1997.
6. Kurt A. Kramer. Identifying Plankton from Grayscale Silhouette Images. Master's thesis, University of South Florida, 2005.
7. Chih-Chung Chang and Chih-Jen Lin. A Library for Support Vector Machines, libsvm. http://www.csie.ntu.edu.tw/-cjlin/libsvm.
8. T. Luo, K. Kramer, D. Goldgof, L. Hall, S. Samson, A. Remsen, and T. Hopkins. Active Learning to Recognize Multiple Types of Plankton. In *International Conference on Pattern Recognition (ICPR)*, Cambridge, UK, August 2004.

Author Index