DATA DISTRIBUTION AND PERFORMANCE OPTIMIZATION MODELS FOR PARALLEL DATA MINING

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Eray Özkural
August, 2013

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Cevdet Aykanat(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. H. Altay Güvenir

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Ömer Morgül

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. İsmail Hakkı Toroslu

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Asst. Dr. Ali Aydın Selçuk

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

DATA DISTRIBUTION AND PERFORMANCE OPTIMIZATION MODELS FOR PARALLEL DATA MINING

Eray Özkural
PhD in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
August, 2013

We have embarked upon a multitude of approaches to improve the efficiency of selected fundamental tasks in data mining. The present thesis is concerned with improving the efficiency of parallel processing methods for large amounts of data. We have devised new parallel frequent itemset mining algorithms that work on both sparse and dense datasets, and 1-D and 2-D parallel algorithms for the all-pairs similarity problem.

Two new parallel frequent itemset mining (FIM) algorithms named NoClique and NoClique2 parallelize our sequential vertical frequent itemset mining algorithm named bitdrill, and uses a method based on graph partitioning by vertex separator (GPVS) to distribute and selectively replicate items. The method operates on a graph where vertices correspond to frequent items and edges correspond to frequent itemsets of size two. We show that partitioning this graph by a vertex separator is sufficient to decide a distribution of the items such that the sub-databases determined by the item distribution can be mined independently. This distribution entails an amount of data replication, which may be reduced by setting appropriate weights to vertices. The data distribution scheme is used in the design of two new parallel frequent itemset mining algorithms. Both algorithms replicate the items that correspond to the separator. NoClique replicates the work induced by the separator and NoClique2 computes the same work collectively. Computational load balancing and minimization of redundant or collective work may be achieved by assigning appropriate load estimates to vertices. The performance is compared to another parallelization that replicates all items, and ParDCI algorithm.

We introduce another parallel FIM method using a variation of item distribution with selective item replication. We extend the GPVS model for parallel FIM we have proposed earlier, by relaxing the condition of independent mining. Instead of finding independently mined item sets, we may minimize the amount of communication and partition the candidates in a fine-grained manner. We introduce a hypergraph partitioning model of the parallel computation where vertices correspond to candidates and hyperedges correspond to items. A load estimate is assigned to each candidate with vertex weights, and item frequencies are given as hyperedge weights. The model is shown to minimize data replication and balance load accurately. We also introduce a re-partitioning model since we can generate only so many levels of candidates at once, using fixed vertices to model previous item distribution/replication. Experiments show that we improve over the higher load imbalance of NoClique2 algorithm for the same problem instances at the cost of additional parallel overhead.

For the all-pairs similarity problem, we extend recent efficient sequential algorithms to a parallel setting, and obtain document-wise and term-wise parallelizations of a fast sequential algorithm, as well as an elegant combination of two algorithms that yield a 2-D distribution of the data. Two effective algorithmic optimizations for the term-wise case are reported that make the term-wise parallelization feasible. These optimizations exploit local pruning and block processing of a number of vectors, in order to decrease communication costs, the number of candidates, and communication/computation imbalance. The correctness of local pruning is proven. Also, a recursive term-wise parallelization is introduced. The performance of the algorithms are shown to be favorable in extensive experiments, as well as the utility of two major optimizations.

Keywords: parallel data mining, graph partitioning by vertex separator, hypergraph partitioning, all pairs similarity, data distribution, data replication.

ÖZET

KOŞUT VERİ MADENCİLİĞİ İÇİN VERİ DAĞITIMI VE BAŞARIM OPTİMİZASYON MODELLERİ

Eray Özkural
Bilgisayar Mühendisliği, Doktora
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ağustos, 2013

Seçilmiş temel veri madenciliği görevlerini iyileştirmek için bir çok yaklaşım üzerinde yoğunlaştık. Şu andaki tez büyük miktardaki veri için paralel işleme metodlarının iyileştirilmesiyle alakalıdır. Hem seyrek hem yoğun verikümeleri için yeni koşut veri madenciliği algoritmaları geliştirdik, ve bütün-çiftler benzerlik problemi için 1-B ve 2-B koşut algoritmalar önerdik.

NoClique ve NoClique2 adında iki yeni koşut veri madenciliği algoritması bitdrill adındaki kendi ardışık dikey sık kalemkümesi madenciliği (SKM) algoritmamızı koşutlaştırmaktadır, ve düğüm ayracı ile çizge bölümleme (DAÇB) kullanan bir metodla kalemleri dağıtmakta ve seçici biçimde yinelemektedir. Metod düğümlerin sık kalemlere ve kenarların iki boyutundaki sık kalem kümelerine karşılık geldiği bir çizge üzerinde çalışmaktadır. Bu çizgenin düğüm ayracının bulunmasının kalem dağıtımı tarafından tespit edilen alt-veritabanlarının bağımsız biçimde işlenmesi için yeterli olduğunu gösterdik. Bu dağıtım uygun ağırlıkların düğümlere verilmesiyle minimize edilen bir veri yinelemesine yol açmaktadır. Veri dağıtımı şeması iki yeni koşut sık kalemkümesi madenciliği algoritmasının tasarımında kullanılmaktadır. Iki algoritma da ayraca karşılık gelen kalemleri yineler. NoClique ayracın sebep olduğu işi yineler ve NoClique2 ayni işi kolektif olarak hesaplar. Hesapsal yük dengeleme ve yinelenen yahut kolektif işin minimizasyonu uygun yük tahminlerinin düğümlere atanmasıyla başarılabilir. Başarım bütün kalemleri yineleyen başka bir koşutlaştırmayla ve ParDCI algoritmasıyla karşılaştırılır.

Seçici kalem yinelemeyle kalem dağıtımını kullanan başka bir koşut SKM algoritması tanıtıyoruz. Daha önce önerdiğimiz koşut SKM için DAÇB modelini, bağımsız madencilik koşulunu gevşetme suretiyle, genişletiyoruz. Bağımsız

keşfedilen kalem kümeleri bulmak yerine, iletişim miktarını minimize edebiliriz ve adayları ince-gözenekli biçimde bölümleyebiliriz. Koşut hesaplamanın düğümlerin adaylara ve hiperkenarların kalemlere karşılık geldiği bir hiperçizge bölümleme modelini öneriyoruz. Her adaya düğüm ağırlıklarıyla bir yük tahmini atanır, ve kalem sıklıkları hiperkenar ağırlıkları olarak atanır. Modelin veri yinelemesini minimize ettiği ve yükleri yüksek kesinlikle dengelediği gösterilir. Aynı zamanda sadece belli bir sayıda seviyenin adaylarını üretebileceğimiz için, önceki kalem dağıtımını temsil eden sabit düğümlerin olduğu bir yeniden bölumleme modeli de tanıtıyoruz. Deneyler NoClique2'nin daha yüksek yük dengesizliğine göre aynı problem örnekleri için, ek koşut fazla hesaplama bedeliyle, hatırı sayılır iyileştirme elde ettiğimizi göstermektedir.

Bütün-çiftler benzerlik problemi için, yakın zamandaki etkin ardışık algoritmaları koşut çerçeveye genişletiyoruz, ve hızlı bir ardışık algoritmanın vektör-başı ve boyut-başı koşutlaştırılmalarını, ve aynı zamanda iki algoritmanın 2-B bir algoritma üreten zarif bir birleşimini elde ediyoruz. Boyut-başı durumu için iki etkin algoritmik optimizasyonun boyut-başı koşutlastırmayı yeterince etkin hale getirdiği gösterilmektedir. Bu optimizasyonlar iletişim bedellerini, aday sayısını ve hesaplama/iletişim dengesizliğini azaltmak için yerel budama ve belli bir sayıdaki vektörun blok işlemesini hedeflemektedir. Yerel budamanın doğruluğu ispatlanır. Ayrıca, özyinelemeli boyut-başı koşutlaştırma sunulur. Geniş deneylerde, algoritmaların başarımının olumlu çıktıgı, ve iki önemli optimizasyonun faydası gösterilmiştir.

Anahtar sözcükler: koşut veri madenciliği, düğüm ayracı ile çizge bölümleme, hiperçizge bölümleme, bütün-çiftler benzerlik, veri dağıtımı, veri yineleme.

Acknowledgement

I acknowledge the following contributions of people who helped with my thesis. Bora Ucar and Cevdet Aykanat came up with the original idea of using graph partitioning by vertex separator for independent mining. I contributed the NoClique algorithm and proved that it would work. I later developed the NoClique2 algorithm in response to reviews. Cevdet Aykanat contributed the hypergraph based graph partitioning by vertex separator algorithm for NoClique2, which was critical for the surprisingly well results that we obtained. The hypergraph partitioning model of frequent itemset mining was based on Aykanat's hypergraph partitioning formulation of graph partitioning by vertex separator problem which we applied to NoClique2. Part of the experimental tests were carried out at the TUBITAK ULAKBIM High Performance Computing Center. We thank Claudio Luchesse for making *ParDCI* available to us. We thank Bart Goethals for providing the benchmark results of FIMI 2004 experiments. The repartitioning model of the hypergraph partitioning approach to frequent itemset mining problem was contributed by Cevdet Aykanat, Ata Turk and Cevdet Aykanat contributed the idea that a two-dimensional algorithm could work for frequent itemset mining, and offered a parallelization based on a mesh network. I later refined that approach to optimize it for non-blocking networks, and also developed the algorithms for it, including the pruning approach. Ata Turk also provided the real world datasets for the parallel all pairs algorithm. Ata also contributed to the theoretical research on that problem, some of which did not make it to the thesis. Cevdet Aykanat carefully reviewed and guided all theoretical research on these problems and contributed the performance analysis frameworks for them, and other miscalleneous bits and pieces that I forgot. Thanks to anonymous reviewers for recommending many improvements. Apologies to others who helped which I may have neglected to mention.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We introduce new parallelization approaches for two fundamental tasks in data mining, that of frequent itemset mining and all pairs similarity, which are the computational basis of several data mining applications. We have embarked upon a multitude of approaches to improve the efficiency of these selected fundamental tasks in data mining. The present thesis is especially concerned with improving the efficiency of parallel processing methods on distributed memory architectures, for large amounts of data for future parallel data mining systems, although the results are applicable to shared memory architectures, as well.

We have devised new parallel frequent itemset mining algorithms that work on both sparse and dense datasets, and 1-D and 2-D parallel algorithms for the all-pairs similarity problem. We propose two new parallel frequent itemset mining (FIM) algorithms named NoClique and NoClique2, the first of which can parallelize any sequential algorithm, and the latter of which parallelizes our own sequential vertical frequent itemset mining algorithm called bitdrill. These algorithms model the parallel FIM task with a graph partitioning by vertex separator (GPVS) model to distribute and selectively replicate items, minimizing data replication. The method operates on a graph where vertices correspond to frequent items and edges correspond to frequent itemsets of size two. We show that partitioning this graph by a vertex separator is sufficient to decide a distribution of the items such that the sub-databases determined by the item distribution can

be mined independently. This distribution entails an amount of data replication, which may be reduced by setting appropriate weights to vertices. The data distribution scheme is used in the design of two new parallel frequent itemset mining algorithms. Both algorithms replicate the items that correspond to the separator. NoClique replicates the work induced by the separator and NoClique2 computes the same work collectively. Computational load balancing and minimization of redundant or collective work may be achieved by assigning appropriate load estimates to vertices. NoClique is a black-box algorithm, and it incurs redundant processing. While it can parallelize any sequential FIM algorithm, as the number of items in the separator grow, so does redundant work. On the other hand, NoClique2 parallelizes a level-wise vertical sequential FIM algorithm we have developed (bitdrill), and the items in the separator correspond to collective work which is mined with a ParDCI like parallelization of bitdrill, and the itemsets are merged with a new frequent itemset merging algorithm which we introduce. NoClique performs very well on sparse datasets, resulting in superlinear speedup for multiple sequential FIM algorithms, but not so well on dense datasets, which is why we had to develop NoClique2. The performance of NoClique2 is compared to another parallelization that replicates all items, and ParDCI algorithm. The experimental results on a linux cluster with 32 single-core compute nodes are consistent and suggest that NoClique2 performs well both on sparse and dense datasets, and compares favorably to state-of-the-art parallel FIM algorithms. The only real shortcoming of this algorithm that we observed was that it sometimes results in large load imbalance.

We additionally introduce another parallel FIM method called Intelligent Candidate and Item Distribution (*ICID*) using a variation of the NoClique2 model for item distribution with selective item replication. We extend the GPVS model for parallel FIM we have proposed earlier, by relaxing the condition of independent mining. Instead of finding independently mined item sets, we may minimize the amount of communication and partition the candidates in a fine-grained manner. We introduce a hypergraph model of the parallel computation where vertices correspond to candidates and hyperedges correspond to items. A load estimate is assigned to each candidate with vertex weights, and item frequencies are given

as hyperedge weights. The model is shown to minimize data replication and balance load accurately. We introduce the *ICID* algorithm which is quite similar to the algorithm of *NoClique2* which first generates the candidates, then applies the hypergraph partitioning model to decide candidate and item distribution, which it uses to redistribute the database that is horizontally partitioned initially, finishing with simultaneous and independent mining of assigned candidates. We also introduce a repartitioning model since we can generate only so many levels of candidates at once, using fixed vertices to model previous item distribution/replication. Experiments show that we improve over the higher load imbalance of NoClique2 algorithm for the same problem instances at the cost of additional parallel overhead.

For the all-pairs similarity problem, we extend recent efficient sequential algorithms to a parallel setting, and obtain document-wise and term-wise parallelizations of a fast sequential algorithm, as well as an elegant combination of two algorithms that yield a 2-D distribution of the data. Two effective algorithmic optimizations for the term-wise case are reported that make the term-wise parallelization feasible. These optimizations exploit local pruning and block processing of a number of vectors, in order to decrease communication costs, the number of candidates, and communication/computation imbalance. The correctness of local pruning is proven. Also, a recursive term-wise parallelization is introduced. The performance of the algorithms are shown to be favorable in extensive experiments, as well as the utility of two major optimizations. In particular, we see promising results up to 256 processors, showing that the term-wise distribution may be quite significant for larger scale, where the typical vector-wise distribution will suffer from the huge bottleneck of full data broadcast required by that distribution.

1.1 Frequent Itemset Mining Problem

1.1.1 Problem Definition

A transaction database consists of a multiset $T = \{X \mid X \subseteq I\}$ of transactions. Each transaction is an itemset and it is drawn from a set I of all items. In practice, the number of items, $|I|$, is in the order of magnitude of 10^3 or more. The number of transactions, $|T|$, is usually larger than 10^5.[1] A pattern (or itemset) is $X \subseteq I$, any subset of I, while the set of all patterns is 2^I. The frequency function $f(T, x) = |\{X \in T \mid x \in X\}|$ computes the number of times a given item $x \in I$ occurs in the transaction database T, and it is extended to itemsets as $f(T, X) = |\{Y \in T \mid X \subseteq Y\}|$ to compute the frequency of a pattern. We use just $f(x)$ or $f(X)$ when T is clear from the context.

Frequent itemset mining (FIM) is the discovery of patterns in a transaction database with a frequency of support threshold ϵ and more. The set of all frequent patterns is $\mathcal{F}(T, \epsilon) = \{X \in 2^I \mid f(T, X) \geq \epsilon\}$. We use just $\mathcal{F}$ when T and ϵ are clear from the context. In our algorithms, two sets require special consideration. $F = \{x \in I \mid f(T, x) \geq \epsilon\}$ is the set of frequent items, and $F_2 = \{X \in \mathcal{F} \mid |X| = 2\}$ is the set of frequent patterns with cardinality 2. In general, F_k is the set of frequent patterns with cardinality k. A significant property of FIM known as downward closure states that subsets of a frequent pattern are frequent, i.e., if $X \in \mathcal{F}(T, \epsilon)$ then $\forall Z \subset X, Z \in \mathcal{F}(T, \epsilon)$ [1].

If all itemsets in $\mathcal{F}$ are enumerated the problem is known as the all FIM problem. Since the size of $\mathcal{F}$ can be large, smaller enumeration problems have been defined such as closed [2] and maximal [3] FIM problems.

[1] These numbers come from the parameters used for the synthetic data generator in [1].

1.1.2 Related work and motivation

FIM comprises the core of several data mining algorithms, such as association rule mining and sequence mining. Frequent pattern discovery usually dominates the running time of these algorithms, therefore much research has been devoted to increasing the efficiency of this task. Since both the data size and the computational costs are large, parallel algorithms have been studied extensively [4, 5, 6, 7, 8, 9, 10, 11, 12]. FIM has become a challenge for parallel computing since it is a complex operation on huge databases requiring efficient and scalable algorithms.

While there are a host of advanced algorithms for parallel FIM, it is desirable to achieve better flexibility and efficiency. We have been inspired by the *Partition* algorithm [13] which divides the database horizontally and merges individual results, as well as Zaki's *Par-Eclat* algorithm [5] which redistributes the database into parts that can be mined independently. Also of immediate interest are the parallelizations of *Apriori* [1], most notably *Candidate-Distribution* [4] which pioneered independent mining. We ask the following questions. Can we design a parallel algorithm that exploits data-parallelism and task-parallelism? Can we find a model to optimize its performance? The present thesis gives an affirmative answer to these questions by introducing an algorithm that divides the database into independently mined parts in a top-down fashion, according to an optimized distribution of the item set.

1.2 All Pairs Similarity Problem

Given a set V of m-dimensional n vectors and a similarity threshold t, the all-pairs similarity problem asks to find all vector pairs with a similarity of t and more. Given the high dimensionality of many real-world problems, such as those arising in data mining and information-retrieval, this task has proven itself to be quite costly in practice, as we are forced to use the brute-force algorithms that have a quadratic running time complexity. Recently, Bayardo et. al [14]

have developed time and memory optimizations to the brute force algorithm of calculating the similarity of each pair in $V \times V$ and filtering them according to whether the similarity exceeds t. We may assume the vectors are in R^m and the similarity function is inner product without much loss of generality.

Two 1-D data distributions are considered: by dimensions (vertical) and by vectors (horizontal). We introduce useful parallelizations for both cases. We have observed that the optimized serial algorithms are suitable for parallelization in this fashion, thus we have designed our algorithms based upon the fastest such algorithm. It turns out that our horizontal algorithms especially attain a good amount of speedup, while the elaborate vertical algorithms can attain a more limited speedup, partially due to limitations in our implementation. Additional contributions to the 1-D vertical distribution includes a local pruning strategy to reduce the number of candidates, a recursive pruning algorithm, and block processing to reduce imbalance. We have also combined the two data distribution strategies to obtain a 2-D parallel algorithm. We also take a look at the performance of a previously proposed family of optimized sequential algorithms and determine which of those optimizations may be beneficial for a distributed memory parallel algorithm design. A performance study compares the performance of the proposed algorithms on small and large real-world datasets.

The rest of the thesis is organized as follows. Chapter 2 gives the background of our target problems, extensive review and analysis of related work. Chapter 3 introduces our GPVS model for parallel FIM task which distributes and selectively replicates items, and proposes two algorithms called *NoClique* and *NoClique2* that apply our model. It also presents an extensive performance study showing both the speedup and quality of the proposed parallel algorithms. Chapter 4 proposes a hypergraph partitioning model that improves upon the load imbalance of the GPVS model, and we also present an elegant algorithm called *ICID* which achieves fine-grain load balancing while eschewing independent mining. We also present a re-partitioning model that allows us to minimize further replication of items, because *ICID* requires multiple iterations to process complex real-world databases. Performance study proves that load imbalance is vastly improved with respect to *NoClique2* and that even without re-partitioning the algorithm surpasses the speedup of *NoClique2* in some cases. We propose new $1-D$ and $2-D$ parallelizations of the all-pairs similarity problem in Chapter 5 that distribute either dimensions, vectors, or both. We introduce two effective optimizations called local pruning, and block processing that address the inefficiency of the algorithm that distributes dimensions. Our extensive experiments show that the performance depends on the dataset. Chapter 6 provides some concluding remarks.

Chapter 2

Background

2.1 Frequent Itemset Mining

2.1.1 Frequent itemset mining algorithms

FIM problem comprises the core of a myriad of data mining tasks [15]. Many mining algorithms append a phase to FIM for extracting useful knowledge from frequent patterns, for instance in association rule mining [16], or their discovery algorithm is remarkably similar or derived from frequent itemset mining such as sequence mining [17] and their derivatives: correlation [18], dependence rule [19], and episode [20] mining. There are several sequential algorithms that have been proposed [15, 1, 21, 22, 13]. With so many algorithms available, a classification is useful. In Zaki's survey paper [23], the large variety of sequential mining algorithms are classified according to their database layout, data structure, search strategy, enumeration, optimizations and number of database scans while Hipp et al. classify them according to search strategy and frequency computation [24].

As the transaction databases are large in both the number of items and transactions, scalability is desirable for FIM algorithms. High performance computing has become an essential element of data mining as very large data is becoming

available in both scientific and business applications. While the sensor data and simulation results accumulate, scientists need better means to analyze them for discovering new knowledge [9, 25].

We must depend on parallel systems to analyze the massive volumes of data in FIM problem [4]. The survey of association rule mining algorithms in [23] not only classifies sequential and parallel mining algorithms according to their design choices but also gives a list of open problems in parallel frequent itemset mining: high dimensionality, large size, data location, data skew, rule discovery, parallel system software, and generalizations of rules. The survey [23] points out the challenges for obtaining good performance as communication minimization, load balancing, suitable data representation, decomposition, and disk I/O minimization. In addition to the requirements of a typical parallel algorithm, a parallel mining algorithm must consider parallelism in disk operations. Zaki identifies three design dimensions: parallel architecture, type of parallelism and load balancing strategy [23]. We refer the reader to Zaki's survey of parallel association rule mining algorithms [23] for a description of some of the algorithms mentioned. Also in [26], the authors analyze the hardware and software requirements of parallel data mining, especially databases, file systems and parallel I/O techniques.

In the following, we review parallelizations of *Apriori* [1] and *Eclat* [22] closely because our work is built upon these two threads of research. Both the *Candidate-Distribution* algorithm (a parallelization of *Apriori* summarized below) and the *Par-Eclat* algorithm are based on the idea of independent mining of database parts. The latter algorithm is especially relevant to our work because it uses the connectivity information in the graph of itemsets with length two. We also point out other related and recent work.

2.1.1.1 Apriori based parallel algorithms

Apriori [1] employs BFS and uses a hash tree structure to count candidate itemsets efficiently. The algorithm generates the set C_k of candidate itemsets (patterns) of length k from the frequent itemsets of length $k-1$ in F_{k-1}. Then, the candidate patterns that have an infrequent sub-pattern are pruned. According to the downward closure lemma, the pruned candidate set contains all frequent itemsets of length k. Following that, the whole transaction database is scanned to determine the set F_k of frequent itemsets among the pruned candidates. This generate and test process is repeated until we have an empty F_k. For higher efficiency, the algorithm uses a hash tree to store candidate item sets (a hash tree has itemsets at the leaves and hash tables at internal nodes [23]).

In [4], the designers of *Apriori* suggest three parallelizations of it. *Count-Distribution* minimizes communication and *Data-Distribution* tries to make use of collective system memory, while *Candidate-Distribution* reduces communication costs by taking task-data dependencies into account and then redistributing data accordingly. Each algorithm parallelizes the iteration which is comprised of a concurrent computation phase and a collective communication phase (except in the largely asynchronous phase of *Candidate-Distribution*).

In *Count-Distribution*, given F_{k-1}, each processor computes all C_k at the beginning of the iteration and scans its local database to determine the local counts. Then, the global counts are computed with a global sum-reduction to all processors. Each processor computes all F_k from global counts.

The objective of *Data-Distribution* is to exploit total system memory better. Each processor generates $|C_k|/n$ candidates. The algorithm is communication intensive since each processor must scan the entire database to determine counts of the candidate sets it owns. As the authors indicate, this algorithm requires fine-grain architectures with low communication-to-computation ratio.

Candidate-Distribution is the most sophisticated of three algorithms as it partitions both data and candidate sets permitting independent mining of parts. This

design was due to the fact that *no* load balancing is done in *Count-Distribution* and *Data Distribution*, a processor has to wait for all other processors at the synchronization step of each iteration. Either of the previous algorithms is used to compute F_{k-1}, an intermediate level in the computation. At the beginning of the kth iteration, the algorithm partitions the set F_{k-1} of frequent itemsets into n parts (on n processors) such that each processor can compute the global counts of its itemsets independently while attaining load balance. At the end of the iteration, the database is redistributed according to the item set partitioning. The partitioning algorithm considers a lexicographical ordering of F_k and F_{k-1}. The itemsets X in F_{k-1} which happen to be the $(k-1)$-length prefixes of itemsets Y in F_k are sufficient to compute the candidates and results of Y [27]. Load balance in partitioning of item sets is achieved by distributing the connected components in a weighted dependency graph which represents candidate generation dependencies among $(k-1)$-length prefixes of F_k. After iteration k, each processor proceeds independently only using pruning information from other processors as it becomes available (a good summary of these algorithms can be found in [23]). Among three algorithms *Count-Distribution* is reported to perform best, in a rather unexpected way since *Candidate-Distribution* is the most advanced design.

2.1.1.2 Parallel algorithms based on Eclat and Clique

Parallel versions of *Eclat* and *Clique* are remarkable in their task distribution strategy which is relevant to our work. Zaki et al. employ two itemset clustering schemes for task parallelism, namely equivalence class clustering and maximal uniform hypergraph clique clustering [5].

Equivalence class clustering uses the same idea as the partitioning in *Candidate-Distribution*. Here we shall demonstrate this scheme with an example from [27]. F_k's in this example have their itemsets represented as *lexicographically ordered strings*. Let $F_3 = \{abc, abd, abe, acd, ace, bcd, bce, bde, cde\}$, $F_4 = \{abcd, abce, abde, acde, bcde\}$, $F_5 = \{abcde\}$. Consider a cluster $\alpha = \{abc, abd, abe\}$ in F_3 with the common prefix ab. Computation of candidates $abcd, abcde, abde, abcde$

with the same prefix depends only on items in α. Depending on this property, each set of items with the same $(k-1)$-length prefix in F_k is identified as a cluster. One of the clusters in this case would be α.

Maximal uniform hypergraph clique clustering obtains a more accurate partitioning by making use of a graph theoretical observation. Let us interpret F_k as a k-uniform hypergraph in which vertices are items and hyper-edges are itemsets of length k. In this hypergraph, the set C of maximal cliques contains all maximal frequent itemsets [5]. In other words, C gives us a good estimate of maximal frequent itemsets, containing all maximal frequent itemsets together with infrequent ones and thus $|C|$ gives us an upper bound on the number of maximal frequent patterns. Clusters are derived in the same way as in equivalence clustering, for each unique $(k-1)$-length prefix in F_k. In the example F_3, the cluster for prefix ab is identified as a set of maximal cliques containing one element $\{abcde\}$.

The number of clusters obtained by the maximal uniform hypergraph clique clustering scheme is greater than the number of processors. These clusters must be assigned to processors so as to maintain load balance. For this purpose, each cluster's load must be weighed. A cluster α is given weight $\binom{|\alpha|}{2}$ which estimates the computational load of frequency mining within the cluster. The clusters are binned to processors with a greedy heuristic.

Vertical representation of a transaction database stores lists of transaction ID's, which are called tidlists, instead of lists of items. It is assumed that F_2 has been computed and tidlists are arbitrarily partitioned in a preprocessing step. Parallel algorithms in [5] are comprised of three phases:

1. itemset clustering and scheduling of clusters among processors,
2. Redistribution of vertical database according to a schedule,
3. Independent computation of frequent patterns.

In all algorithms, F_2 is used for partitioning so that redistribution can be made as soon as possible. Independent mining is performed by either a BFS or hybrid DFS/BFS search strategy.

Zaki et al. [5] underline the advantages of their algorithms as distribution of data, decoupling of the processors in the beginning, vertical database layout, and fast intersections avoiding structure overhead. In the experiments, it is seen that the more advanced maximal clique clustering does in fact improve upon equivalence class clustering by providing more exact load balancing information. An important contribution of [5] is the application of itemset clustering to determine independent mining sub-tasks.

Candidate-Distribution algorithm [4, 27] and *Par-Eclat* [5] are built on the idea of independent mining of database parts. Both algorithms mine up to a level using a simple parallelization and then redistribute the data such that the processors mine independently. *Par-Eclat* algorithm is especially relevant to our work because it uses the connectivity information in the graph of itemsets with length two (however it could have been any level using a hypergraph). It distributes candidate itemsets by clustering maximal cliques.

2.1.2 Other studies and remarks

A tight upper bound on the number of maximal candidate patterns given F_k is presented in [28]. It is also shown experimentally that the estimates are fairly accurate in mining artificial data sets. It is suggested that the results may be used in the optimization of mining algorithms. This theoretical work may be useful for improving load balance in parallel mining algorithms.

Some relevant algorithms are as follows. *ParDCI* [10, 29] is a practical parallelization of *DCI* [30] which mines using a level-wise method up to a level and replicates the entire database when it fits into memory; good speedups are reported on a cluster of SMP's. A recent theoretical paper on the closed FIM problem [11] partitions the search space so that each part can be mined independently, and thus in parallel. A parallel association rule mining algorithm is introduced in [31] which replicates a novel layout of the database on all processors. In [7], a parallel implementation of *FP-Growth* is presented and good speedups are reported on a distributed collective-memory SGI Origin machine with very

sparse databases (there are hundreds of thousands of items). Another parallelization of *FP-Growth* distributes $N-1$ projections of the database for N items (a technique which was first described in [15]), and reports experiments on a PC cluster [8]. A recent parallelization of *FP-Growth* uses the novel method of selective sampling to improve load balancing [32]. *Intelligent-Data-Distribution* and *Hybrid-Distribution* are scalable parallel association rule algorithms tested on the Cray T3D [9]. In [33], parallel tree-projection-based sequence mining algorithms for data and task-parallel formulations have been introduced, the latter of which uses graph partitioning. In [34], a distributed FIM algorithm suitable for large distributed systems with scarce communication resources is presented. Recently, a distributed FP-Growth implementation has been used for query recommendation [12]. Parallel reconfigurable computing architectures have also been explored in the context of frequent itemset mining [35, 36].

Also of interest to our work is applications of hypergraph partitioning to problems with complex task-data dependencies. A very good example of such a problem is direct volume rendering. Cambazoglu and Aykanat model the image-space parallelization of this problem in [37], where pixel block rendering tasks correspond to vertices and hyperedges correspond to object-space data (cell clusters). They also propose a particularly interesting remapping model that provides an incremental algorithm which models the past mapping using fixed processor vertices, by which we are inspired in the redistribution model of our hypergraph partitioning approach to FIM problem. Aykanat et. al also propose a view-dependent parallelization of the problem in object space, this time presenting a graph model of the parallel task in [38], where the vertices correspond to cells and edges correspond to shared faces.

2.2 All Pairs Similarity

2.2.1 Problem definition

Following a similar terminology to [14], let $V = \{v_1, v_2, v_3, ..., v_n\}$ be the set of sparse input vectors in R^m. Let t be the similarity threshold. Let a sparse vector x be made up of m components $x[i]$, where some $x[i] = 0$; such a sparse vector can be represented by a list of pairs $[(i, x[i])]$ in which only non-zero components are stored. Let $|x|$ be the number of non-zero components in the vector, that is the length of its list representation. Let $||x||$ be the vector's magnitude. Let also $size(V) = \sum_{v \in V} |v|$ be the number of non-zero values in V. Each vector v_i is made up of components per dimension d, where the vector's dth component is denoted as $v_i[d]$. The similarity function is defined as the summation of input values from similarity among individual components: $sim(x, y) = \sum_i sim(x[i], y[i])$. Another accumulation function instead of summation may be used (for instance any other binary operation which has the same algebraic properties), however summation is enough for many purposes. The problem is to find the set of all matches $M = \{(v_i, v_j) \mid v_i \in V \wedge v_i \in V \wedge i \neq j \wedge sim(v_i, v_j) \geq t\}$.

Without much loss of generality, we assume that input vectors are normalized (for all $x \in V, ||x|| = 1$), and for vectors x and y, $sim(x, y)$ function is the dot-product function $dot(x, y) = \sum_i x[i].y[i]$, that is $sim(x[i], y[i]) = x[i].y[i]$. The algorithms can be easily generalized to other similarity functions which are composed from similarities $sim(x[i], y[i])$ across individual dimensions.

The input dataset V may also be interpreted as a data matrix D where row i is vector v_i. In this case, we may represent similarities by the similarity matrix $S = D.D^T$ where $S_{ij} = dot(v_i, v_j)$ obviously, and we find the set of matches $M = \{(i, j) \mid S_{ij} \geq t\}$. More naturally, we may interpret the output as a match matrix M that is defined as:

$$M'_{ij} = \begin{cases} 0 & \text{if } S_{ij} < t, \\ S_{ij} & \text{if } S_{ij} \geq t \end{cases} \tag{2.1}$$

The output set of matches M may be considered to define an undirected similarity graph $G_S(V, t) = (V, M)$. In this case an edge $u \leftrightarrow v$ denotes a similarity relation between vectors u and v; the edge weight $w(u, v) = u.v$.

2.2.2 Applications

An all pairs similarity algorithm may be viewed as a computational kernel for several tasks in data mining and information retrieval domains. In data mining and machine learning, the similarity graph may be supplied as input to efficient graph transduction [39, 40], graph clustering algorithms [41] and near-duplicate detection (by using a high threshold to filter edges). Obviously, once a similarity graph is computed, classical k-means [42, 43] or k-nn algorithms [44, 45], which are widely used in data mining due to their effectiveness in low number of dimensions, may be adapted to use the graph instead of making the geometric calculations directly over input vectors. As frequent itemset mining may be viewed as the costly phase of association rule mining class of algorithms; likewise, the graph similarity problem may be viewed as the costly phase of several classification, transduction, and clustering algorithms.

Calculating the similarity graph may be alternately viewed as capturing the essential geometry of (the similarities in) the dataset, on which any number of computational geometry algorithms may be run. This is basically what a classification or clustering algorithm does given similarities in the data: the algorithm tries to find geometric distinctions, either determining a class boundary for classification, or identification of clusters by grouping similar points according to the similarity geometry. Note also that with an adequate similarity threshold, we can obtain a connected graph and therefore approximate *all* similarities in the dataset.

Constructing the similarity graph also has the unique advantage in that it can be re-used later for additional data mining tasks. For instance, one application can make a hierarchical clustering of the data, and another one can use it for transduction. Basically, we think that any data mining task that has a *geometric*

interpretation can use the similarity graph as input successfully. Therefore, we anticipate that the parallel similarity graph construction will be a staple of future parallel data mining systems.

2.2.3 k-nearest neighbors problem

The problem of constructing a similarity graph can be contrasted with k-nearest neighbors problem, which is a slightly harder problem but can be solved approximately using a distance threshold. Our use of the dot-product between two vectors should not be misleading either, as that corresponds to range search in a corresponding metric space, to emphasize the close relation between these problems. At any rate, some of the same approaches can be adapted to similarity graph construction, therefore we should take them into account. Especially, note that most of the difficulties with nearest neighbor search carry over to our problem.

Due to the curse of dimensionality [46], the brute-force algorithm of nearest neighbor search is quite difficult to improve upon [47]. In practice, there are no advanced geometric data structures that will give us algorithmic shortcuts [48, 49]. In the general setting of metric spaces, the nearest neighbor problem is non-trivial and data structures are not very effective for high dimensionality [50]. This implies that we cannot rely on space partitioning or metric data structures that work well in low number of dimensions, although of course, non-trivial extensions of those methods may prove to be effective such as combining dimensionality reduction with geometric data structures.

2.2.4 Related sequential algorithms

2.2.4.1 Sequential knn algorithms

Some popular approaches to solving the nearest neighbor problem may be summarized as geometric data structures such as R-Tree[51]; VP-Tree [52], GNAT [53]

and M-Tree [54] for general metric spaces, pivot-based algorithms [55, 56], random projections for ϵ-approximate solutions to the knn problem [57], combining random projections and rank aggregation for approximation [58], locally sensitive hashing [59, 60, 61], and other data structures and algorithms for approximations [62, 63]. An algorithm related to our area of interest detects duplicates by using an inverted index [64]. Space-filling curves have also been applied to the knn problem [65, 66, 67].

Space-partitioning approaches usually do not work well for very high-dimensional data due to the curse of dimensionality, a thorough treatment of which is available in [47]. Weber et. al quantify in that article lower bounds on the average performance of nearest neighbor search for space and data partitioning assuming uniformly distributed points, which show that for space partitioning like k-d trees, the expected NN-distance grows with increasing dimensionality, rendering such methods ineffective for high-dimensional data (full scan needed when $d > 60$), and for data-partitioning the number of blocks that have to be inspected increase rapidly with increasing number of dimensions, for both rectangular (full scan is faster when $d > 26$) and spherical bounding regions (full scan when $d > 45$), and they also generalize their results to any clustering scheme that uses convex clusters, not just these. Their conclusion is that in high-dimensional data, the partitioning methods all degenerate to sequential search, in uniformly distributed data. We emphasize that their results imply that trivial geometric partitions of the data using hyperplanes or hyperspheres are mostly ineffective in very high-dimensional data, although they can in some cases work well for datasets with limited dimensionality or different distribution. Weber et. al for this reason propose the VA-file, which approximates vectors using bitstrings [47] and improves upon sequential scan.

In general, it seems that for solving proximity problems exactly in very high-dimensional datasets, techniques that prune candidates work well. Kulkarni and Orlandic, on the contrary, successfully use a data clustering method to optimize knn search in databases, which the authors show to be better than sequential scan and VA-file up to 100 dimensions on random datasets and 56 dimensions on real-world datasets [68], although it is impossible to know the true efficiency of

these algorithms proposed by database researchers unless they are compared to fast in-memory algorithms since disk access time dominates the running time of algorithms that work on secondary storage. Also, such approaches do not usually scale up to very high number of dimensions.

Note that there are asymptotically optimal nearest neighbor algorithms in the literature. Vaidya introduces an asymptotically optimal algorithm for the all nearest neighbors problem which has $O(n \log n)$ time complexity [69]. The same algorithm solves k-nearest neighbors problem in $O(n \log n + kn \log k)$ time, while Callahan and Korasaraju propose an optimal k nearest neighbors algorithm which runs in $O(n \log n + kn)$ time [70]. It is not immediately obvious why there are no experiments measuring the real-world performance of these optimal algorithms, however, it is conceivable that they may not have been practical for high-dimensional datasets, or it may have been considered that they require large constant factors.

We refer the reader to Chavez's survey of search methods in metric spaces [71] for more information on the myriad algorithms. Chavez identifies three kinds of search algorithms for metric spaces: pivot-based algorithms, range coarsening algorithms, and compact partitioning algorithms, and he emphasizes that the search time of exact algorithms grow with intrinsic dimensionality of the metric space, which also increases the search radius, and thus makes it harder to compete with brute-force algorithms. As we have seen, similar problems also plague search algorithms in Euclidian spaces. For these reasons, researchers in recent years have turned to practical optimizations over brute-force algorithms, which we shall now examine briefly with a good example.

2.2.4.2 Practical sequential similarity search

In Bayardo et. al [14], the authors propose three main algorithms which embody a number of heuristic improvements over the quadratic brute force all-pairs similarity algorithm. These algorithms are summarized below. In the algorithms, each vector x has components with weights $x[i]$, there are m dimensions (or features)

Algorithm 1 *All-Pairs-0*(V, t)

$M \leftarrow \emptyset$
$I \leftarrow$ *Make-Sparse-Matrix*(m, n)
for all $v_i \in V$ **do**
  $M \leftarrow M \cup$ *Find-Matches-0*(v_i, I, t)
  for all $v_i[j]$ where $v_i[j] > 0$ **do**
    $I_{ji} \leftarrow v_i[j]$
return M

Algorithm 2 *Find-Matches-0*(x, I, t)

$A \leftarrow$ *Make-HashTable()*
for all $(i, x[i]) \in x$ where $x[i] \neq 0$ **do**
  for all $(y, y[i]) \in I_i$ **do**
    $A[y] \leftarrow A[y] + x[i].y[i]$
return $\{(y, A[y]) \mid A[y] \geq t\}$

numbered from 1 to m, $maxweight_i(V)$ is the maximum weight in dimension i of the entire dataset V, and $maxweight(x)$ is the maximum weight in a vector x, following the notation in their paper.

all-pairs-0 This is equivalent to the brute force algorithm, with the additional on-the-fly construction of an inverted index as each vector is matched and indexed in turn. The calculation of the dot-product scores are achieved by consulting the inverted index. Thus each vector is compared to all the previous vectors that have been indexed, and then the vector itself is added to the index. This algorithm is thus slower than the brute force algorithm. In the matching of a new vector x, the algorithm uses a hash table A to store the weights of candidates to match against x, since the vectors are sparse. The pseudocode for all-pairs-0 is given in Algorithm 1 and Algorithm 2.

all-pairs-1 This algorithm orders the dimensions in the order of decreasing number of non-zeroes. It corresponds to an important optimization that we call "partial indexing" which works as follows. In preprocessing, we calculate $maxweight_i(V)$ for each dimension. This allows us to calculate an upper bound for the dot-product of a vector x with any vector in V: $\forall y \in V$ $x.y \leq \sum_i x[i].maxweight_i(V)$. Using this upper bound it is possible to

avoid indexing the most dense dimensions by calculating a partial upper bound b while processing the components of new vector x for indexing. Remember that we are processing the components in a certain order (decreasing number of non-zeroes of dimensions in V). The components are added to the inverted index only when the partial upper bound b exceeds t, the initial components that have small b are not indexed at all, they are kept as a partial vector x'. Indexing as such ensures that all admissible candidate pairs are generated. The dot-product is fixed by adding the dot-products of the partial x''s later on.

all-pairs-2 This algorithm affords three optimizations over all-pairs-1.

Minsize optimization: This optimization aims to prune candidate vectors with few components. We know that for a vector x, for all matches y, $x.y \geq t$. If the input vectors are normalized, then each component can be at most 1: $x.y < maxweight(x).|y|$. Two inequalities entail that $|y| \geq t/maxweight(x)$. Let the quantity on the right be called *minsize*'. Minsize optimization requires the vectors to be ordered in order of increasing $maxweight(x)$, thus decreasing *minsize*. If ordered such and the input vectors are normalized, during matching a new vector x, the minimum size of a candidate vector y that x can be matched against is $t/maxweight(x)$. If the candidates in the inverted index that are smaller than *minsize* are pruned when matching a new vector, this will hold true for all the subsequent vectors since *minsize* for subsequent vectors cannot be greater. The minsize optimization does not prune a lot of candidates, but it may be effective since there may be a lot of very small vectors. It is suggested that all-pairs-2 prunes only vectors in the beginning of the inverted list, which is easy to implement using dynamically sized arrays.

Remscore optimization: This optimization calculates a maximum remaining score (remscore) while processing the components of a vector x during matching, using $maxweight_i(V)$ function. When remscore drops below t the algorithm switches to a strategy that avoids adding any new candidates to the candidate map, while continuing to update the candidates already in the map. This avoids calculation of scores for candidates that cannot match.

Remscore is initialized as $\sum_i x[i].maxweight_i(V)$ and as each component i is processed its contribution to the upper bound $x[i].maxweight_i(V)$ is subtracted from the upper bound. And while calculating the scores in the candidate map, the aforementioned conditional is executed. While this seems to be an excellent optimization, in the real-world data we have seen it has only inflated the running time, because not the calculation of remscore but the conditional reasoning is too expensive within the main loop of matching algorithm.

Upperbound optimization: While fixing the scores in the candidate map with dot-products of partial vectors (parts of vectors that are not indexed), we can avoid the dot-product if the following upper bound is not enough to make the score exceed t: $min(|y'|, |x|).maxweight(x).maxweight(y')$ which is to say that each scalar product in an inner product cannot be more than the product of the maximum values in either vector, and only non-zero components contribute to the inner product. While this too seems to be a nice optimization, it suffers from using conditionals in an otherwise efficient code as the partial vectors tend to be short.

2.2.4.3 Analysis of all-pairs-0

All-pairs-0 maintains an inverted index I, which stores an inverted list for each of m dimensions in the dataset, such that after all the matches are found, for a vector v_i and for all $v_i[j]$, the inverted index I stores $v_i[j]$, that is $I_{ji} = v_i[j]$.

If the inverted index I is interpreted as a matrix, the rows I_j of the inverted index are the dimensions in the dataset, and I is merely the transposition of the input matrix D, $I = D^T$. Algorithm all-pairs-0 performs $\sum_{d=1}^{m} \binom{|I_d|}{2}$ floating-point multiplications, dominating the running time complexity, therefore each dimension d contributes $\binom{|I_d|}{2} = O(|I_d|^2)$ multiplications.

Since in practice there are usually a few dense dimensions, the running time complexity is expected to be quadratic in n for real-world datasets.

2.2.5 Related parallel algorithms

There are only a few relevant studies on efficient parallelization of the all pairs similarity problem in the literature that we have been able to detect.

Lin [72] parallelizes the all-pairs similarity problem comparing parallelizations of both the brute force algorithm that uses no intermediate data structures and two algorithms that use an inverted index of the data, one horizontal and one vertical parallelization (called Posting Queries and Postings Cartesian Queries algorithms), implemented with the map/reduce framework Hadoop. The algorithm is cast in an information retrieval context where documents are vectors and terms are dimensions. The experiments are quite comprehensive and utilize realistic life sciences datasets. The study in question also compares the performance of three approximate solutions: limiting the number of accumulators, considering only top n terms in a document, and omitting terms above a document frequency threshold; their results show that significant performance gains can be obtained from approximate solutions at acceptable loss of precision. Therefore, Lin suggests that parallelizing the exact algorithms easily carry over to more efficient inexact algorithms. However, there is a slight drawback of this careful study, as the use of Java language may have caused significant performance loss in the sequential algorithms, making the job of parallelization easier, as for 90 thousand documents, their sequential algorithm takes on the order of hundreds of minutes on a cluster system. Lin does mention that the code is not optimized and run on a shared, virtualized environment. In our experience, shared environments are not suitable for working on memory and communication intensive problems such as those in information retrieval and data mining. Thus, we are looking forward to the repetition of the said experiments on a dedicated parallel computer with a more appropriate high-performance implementation. This study is also important in that the author correctly observes the influence of the Zipf-like distribution of terms on parallel performance.

Recently Awekar et. al [73] introduced a task parallelization of the all pairs similarity problem, sharing a read-only inverted index of the entire dataset. The

authors use a fast sequential algorithm which is very similar to our all-pairs-0-array, which we also found to be the best sequential algorithm, and thus make adequate speedup measurements. The authors test three load balancing strategies, namely block partitioning, round-robin partitioning, and dynamic partitioning on high-dimensional sparse data-sets with a power law distribution of vector sizes. Their experiments are executed on up to 8 processors for large real-world datasets, on both a shared-memory architecture and a multi-processor system. The speedups on the multi-processor system turn out to be superior to the shared memory system as cache-thrashing and memory-bandwidth limitation prevents near-ideal performance for larger number of processors on shared-memory systems. In this study [73], however, there is a major shortcoming as the index construction and replication costs were not taken into account in the experiments, which raises doubts as to how much time is needed for broadcasting such large datasets (e.g., Orkut dataset has 223 million non-zeroes), as the replication of the entire inverted index would be a bottleneck for high number of processors. Therefore, the replicated index algorithm should be taken with a grain of salt, as well as any parallel algorithm that replicates the entire dataset, since the size of the inverted index is the same as the size of the dataset. At any rate, near-ideal speedup on up to 8 processors is not surprising as our vector-wise parallelization shows similar performance, as will be seen.

Following are parallelizations of related problems. Plaku and Kavraki propose a distributed, message-passing algorithms for constructing knn graphs of large point sets with arbitrary distance metric [74]. They can use any local knn data structure for faster queries (such as a metric tree), which must be built once the points are distributed to processors. In addition to this, they can exploit the triangle inequality of metric function and this information can be used to construct local queries using the metric data structure as well as pruning distributed queries, by representing the bounding hyperspheres of points on other processors. The dimensionality of their datasets increases to non-trivial numbers (up to 1001), and their speed-up results on 100 processors are quite encouraging. We think that their method might be applied to our work as well in the future, to optimize our horizontal parallel algorithms, however the effectiveness of their

approach on very high-dimensional datasets as we are using remains to be seen, as no sort of space partitioning usually works well for very high-dimensional datasets due to the curse of dimensionality. However, it is conceivable that the methods of Plaku and Kavraki could be used in hybrid approaches to deal with much higher dimensionality. A shortcoming of this paper is that it does not discuss the partitioning of the point set, any partition is assumed.

Alsabti et al. [75] parallelize all pairs similarity search with a k-d tree variant using two space-partitioning methods based on quantiles and workload; they find that their method works well for 12-d randomly generated points on up to 16 processors. Their workload based partitioning scales better than quantile based partitioning, and is comparable for uniform and gaussian distributions. Aparício et. al [76] use a three-level parallelization of knn problem at the Grid, MPI and shared memory levels and integrate all three to optimize performance. An interesting paper proposes a parallel clustering algorithm which partitions a similarity graph, constructs minimum spanning trees for each subgraph and then merges the minimum spanning trees, which is then used to identify clusters [77]; this algorithm can be applied to the output of our algorithms. Schneider [78] evaluates four parallel join algorithms for distributed memory parallel computers from a database perspective. Vernica et al. [79] propose a three-stage map/reduce based approach to calculate set similarity joins and report results using Hadoop; they do consider the self-join case.

Callahan and Kosaroj [70, 80] examine the well-separated pair decomposition of a point set in Euclidian space, which decomposes the set of all pairs in a point set into pairs of sets with the constraint of well-separation (defined in a certain geometric sense), wherein each pair is uniquely represented by a pair of point sets in the decomposition. Using their decomposition, they also obtain an asymptotically optimal parallel knn algorithm which has $O(\log^2 n)$ total parallel time on $O(n)$ processors with the CREW PRAM model. The real-world applicability of this wonderfully efficient algorithm remains to be seen, however. In our initial inspection, we have seen their splitting logic may be somewhat problematic in text data sets where each co-ordinate corresponds to the frequency of a term. It seems that one way such space decomposition based algorithms may escape the

curse of dimensionality is that the decomposition is far from random, and that the distribution is not uniform in real-world datasets, although one may still expect that the approach might break down in very high-dimensional datasets as their approach is conceptually similar to well known k-d tree construction algorithms that fail in high-dimensional datasets.

Chapter 3

Parallel Frequent Itemset Mining with Selective Item Replication

We introduce our transaction database distribution scheme for parallel frequent itemset mining problem and its theoretical analysis in Section 3.1, while Section 3.2 introduces the *NoClique* and *NoClique2* parallel algorithms which identify lack of cliques among sets of itemsets. Section 3.3 presents an extensive performance study of the proposed algorithms.

3.1 Transaction Database Distribution

In this section, we describe our theoretical contributions which will be developed into a parallel algorithm in Section 3.2. We make heavy use of the GPVS problem, which is briefly explained in the following.

The GPVS problem is to find a minimum weighted vertex separator V_s, removal of which decomposes a graph into components with roughly equal weights [81]. Let $G = (V, E)$ be a graph where $w(u)$ is the weight of vertex u. Let $w(U) = \sum_{u \in U} w(u)$ be the weight of a vertex set U. Let $Adj(u)$ denote the set of vertices that are adjacent to u, i.e., $Adj(u) = \{v|(u, v) \in E\}$. This

operator can be extended to vertex sets by letting $Adj(U) = \bigcup_{u \in U} Adj(u) - U$.

Definition 1 (n-way GPVS)**.** $\Pi_{VS}(G) = \{V_1, V_2, \ldots, V_n : V_s\}$ *is a partition of the vertex set* V *into* $n+1$ *subsets* $V_1, V_2, \ldots, V_n$ *and* V_s *such that for all* $1 \leq i < j \leq n$ $Adj(V_i) \cap V_j = \emptyset$ *(i.e.,* $Adj(V_i) \subseteq V_s$*). The partitioning objective is to minimize* $w(V_s)$*. The partitioning constraint is, for all* $1 \leq i \leq n$*,* $w(V_i) \cong [w(V) - w(V_s)]/n$ *(parts have roughly the same weight).*

The problem is NP-complete [82, ND 25 Minimum b-vertex separator]. A separator V_s is said to be minimal if there is no subset of V_s that is also a separator. The two-way GPVS will be denoted as $\Pi_{VS}(G) = \{A, B : S\}$.

We introduce a distribution method that can be used to divide the FIM task in a top-down fashion. The method operates on the graph G_{F_2} which is defined as follows.

Definition 2. $G_{F_2}(T, \epsilon) = (F, F_2)$ *is an undirected graph in which each vertex* $u \in F$ *is a frequent item and each edge* $\{u, v\} \in F_2$ *is a frequent pattern of length two, for a given database* T *and support threshold* ϵ*. The parameters* T *and* ϵ *will be dropped when they are clear from the context.*

We decode a two-way GPVS of the G_{F_2} graph as a two-way distribution of the transaction database such that the two sub-databases obtained can be mined independently and therefore utilized for concurrency. In order for this property to hold, there is an amount of replication dictated by the vertex separator of G_{F_2}, which corresponds to the partitioning objective of GPVS. In the following, we first present the optimization aspects of our transaction database distribution technique. Then, we expound on our GPVS model for two-way transaction database distribution. Afterwards, we discuss minimization of data replication, followed by minimization of collective work and load balancing in the GPVS model. We then extend the two-way distribution scheme to n-way (for n processors). Last, we show that our method is applicable to maximal and closed FIM problems.

3.1.1 Optimizing parallel frequent itemset discovery

Our objective of transaction database distribution is to divide a transaction database such that each sub-database can be mined independently, while not inflating the data prohibitively and keeping the computational load balanced across sub-databases. Once such a distribution is obtained, a coarse-grain parallel frequent itemset mining algorithm similar to *Par-Eclat* can be designed. *Par-Eclat* consists of a redistribution phase and a following local mining phase with no communication [5]. We present two algorithms: *NoClique* features completely independent mining with no communication just like *Par-Eclat*, while *NoClique2* has a collective phase in which the running time is minimized and the rest of mining is independent. Since some data mining tasks on the sub-databases are performed independently in either algorithm, our method may be classified as a data-parallel algorithm that adopts input data partitioning with replication. This input data partitioning induces a task partitioning according to the owner-computes rule [83, Sec. 3.2.2], which states that the process assigned a particular data item is responsible for all computation associated with it.

We show that GPVS on G_{F_2} is sufficient to designate such a distribution on the transaction database. Our work assumes that G_{F_2} is sparse, because GPVS may not be feasible on dense graphs. Note that a sparse G_{F_2} does not necessarily require the input database to be sparse.

We may begin formulating a problem for the coarse-grain data-parallel frequent itemset mining algorithm as follows: Let database T contain a smaller database T_i. T_i is a sub-database of database T if and only if, for every transaction $X \in T_i$, there is a distinct transaction $Y \in T$ such that $X \subseteq Y$ (recall that T and T_i are multisets). We will denote this ordering relation with $T_i \prec T$. The input database T is distributed to a number of processors such that each processor has a sub-database of the original transaction database. We denote this distribution by $D(T) = \{T_i | T_i \prec T\}$, possibly with replication. Also, we require the union of frequent patterns discovered in individual processors to be the set of frequent patterns of the entire data, i.e., $\mathcal{F}(T, \epsilon) = \bigcup_{T_i \in D(T)} \mathcal{F}(T_i, \epsilon)$. We call this the independent mining condition for a distribution $D(T)$.

In the following optimization problem, $w(\cdot)$ is any sensible cost measure that relates to mining a database, e.g., computational work, data size:

$$\text{minimize} \quad \left(\sum_{T_i \in D(T)} w(T_i) \right) - w(T) \tag{3.1}$$

$$\text{subject to} \quad T_i \prec T, \text{for all } T_i \in D(T) \tag{3.2}$$

$$\mathcal{F}(T, \epsilon) = \bigcup_{T_i \in D(T)} \mathcal{F}(T_i, \epsilon) \tag{3.3}$$

$$w(T_i)\text{'s are approximately equal} \tag{3.4}$$

The objective in Equation 3.1 seeks to minimize the total amount of redundancy that the distribution $D(T)$ entails. We subtract the cost of the entire database from the sum of costs of distributed sub-databases T_i's to denote this. Equation 3.2 is the distribution condition which states that the transaction database is distributed in any fashion, e.g., transaction-wise, item-wise or hybrid. Equation 3.3 is the independent mining condition, which ensures that independent mining of the sub-databases yields the frequent patterns of the entire database. The balancing condition (Equation 3.4) ensures that all processors share the cost fairly. At this stage, we do not explicitly state whether we are minimizing data redundancy or parallel overhead. However, some amount of data replication is often necessary for the independent mining condition to hold.

We will now expose our particular item redistribution scheme using information in frequent itemsets of length two, which can be easily computed in parallel like in the design of *Par-Eclat* [5]. First, we will show how we can satisfy the distribution and independent mining conditions by showing a two-way item distribution. We will then analyze the objective and the balancing condition, explaining how we can assign weights and achieve load balance so that it becomes an acceptable solution to the coarse-grain parallel FIM problem.

3.1.2 Two-way item-wise transaction database distribution

G_{F_2} is relatively easy to compute with respect to the complexity of the whole mining task, and its computation is amenable to efficient parallelization. It contains information that can be used to predict computational properties. For instance, the maximal cliques in G_{F_2} give us potentially maximal patterns [5], which in turn can be used to achieve task parallelism. Our data decomposition method, on the other hand, does not require finding maximal cliques. Instead, we use the GPVS of G_{F_2}, which allows us to define independent mining on the transaction database by finding a particular distribution of the item set I. Our item distribution identifies the absence of cliques across two sets of items rather than enumerating all cliques as in [5].

We will start by observing the similarity of GPVS objectives to ours. It turns out that we can use a GPVS of G_{F_2} to satisfy the independent mining conditions and to optimize parallelism at the same time. FIM task can be decomposed into mining two item-wise projections of the transaction database using GPVS. We use the projection operator π to explicitly show the vertical projections.

Definition 3. *A transaction database projected from T over a set of items X is $\pi_X(T) = \{Y \cap X \mid Y \in T\}$ where Y is a transaction in T.*

Recall that two-way GPVS is denoted as $\Pi_{VS}(G) = \{A, B : S\}$ where S is the vertex separator; and A and B are vertex parts. GPVS of G_{F_2} corresponds to a certain two-way distribution $\{A \cup S,\ B \cup S\}$ of the itemset I. This distribution induces a two-way transaction set distribution as follows.

Definition 4. *A two-way transaction database distribution $D(T) = \{T_1, T_2\}$ is induced by $\Pi_{VS}(G_{F_2}) = \{A, B : S\}$, where $T_1 = \pi_{A \cup S}(T)$ and $T_2 = \pi_{B \cup S}(T)$.*

We require S to be a minimal separator. If S were not minimal, since the cost induced by the separator is included in both projections, removing a vertex from the separator would decrease the parallel cost. For that reason, it is better

to choose a minimal separator, in case the GPVS heuristic does not find one. Figure 3.1 depicts a sample transaction database and its G_{F_2} graph. Π_{VS} of this graph and the transaction database distribution $D(T)$ induced by Π_{VS} is illustrated in Fig. 3.2. In the following text, we show that mining the database parts separately results in complete FIM of the original transaction database T satisfying Equation 3.3.

Transaction	a	b	c	d	e	f	g	h	i
$t_1 = \{b, c, f, g\}$		×	×			×	×		
$t_2 = \{c, g\}$			×				×		
$t_3 = \{b, h, i\}$		×						×	×
$t_4 = \{b, e\}$		×			×				
$t_5 = \{a, d, g, h\}$	×			×			×	×	
$t_6 = \{d, e\}$				×	×				
$t_7 = \{b, c, e, f\}$		×	×		×	×			
$t_8 = \{a, b, c, f\}$	×	×	×			×			
$t_9 = \{b, c, d, h, i\}$		×	×	×				×	×
$t_{10} = \{b, c, e, g\}$		×	×		×		×		
$t_{11} = \{a, d, e, g\}$	×			×	×		×		
$t_{12} = \{d, h\}$				×				×	
$t_{13} = \{a, d, e, h\}$	×			×	×			×	
$t_{14} = \{b, e\}$		×			×				
$t_{15} = \{a, e, g\}$	×				×		×		

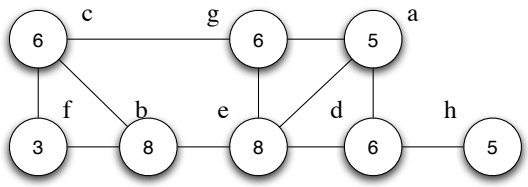


Figure 3.1: *Top:* A sample database T with 15 transactions and 9 items. *Bottom:* G_{F_2} graph of T with a support threshold of 3. The vertices are labeled with the number of times an item occurs in the database.

Lemma 1. *If there is a frequent pattern P in T, then there is a corresponding clique in G_{F_2}, with vertices corresponding to items in P.*

Proof. Due to the downward closure property, a pattern P can be frequent if and only if all subsets of the pattern are frequent patterns. A frequent pattern

$P \subseteq F$ contains $\binom{|P|}{2}$ subsets (sub-patterns) with cardinality 2. That is, for all $u, v \in P, \{u, v\}$ is a frequent pattern. By the definition of G_{F_2}, each frequent pattern of length 2 is an edge in G_{F_2}; hence for all $u, v \in P, \{u, v\} \in F_2$ showing the existence of a corresponding clique in G_{F_2}. □

Lemma 2 (No Clique)**.** *There is no frequent pattern with items in both A and B parts of $\Pi_{VS} = \{A, B : S\}$ of G_{F_2}.*

Proof. This follows from Lemma 1 that there is no clique in G_{F_2} with vertices in both A and B of $\Pi_{VS} = \{A, B : S\}$ of G_{F_2}. See Fig. 3.3 for an illustration of the proof. □

Corollary 1. *There is no frequent pattern with items in both A and B parts of $\Pi_{VS} = \{A, B : S\}$ of G_{F_2}.*

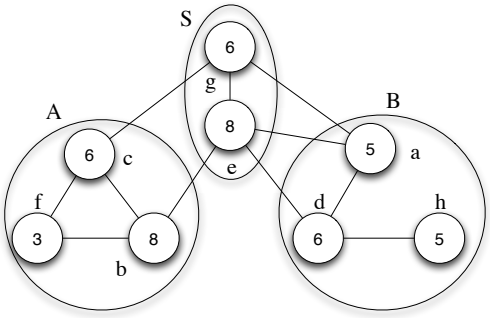
Proof. Since there can be no clique in G_{F_2} that contains items in both A and B, there can be no such frequent pattern. □

Theorem 1 (Independent Mining)**.** *Independent discovery of frequent patterns in projected databases $T_1 = \pi_{A \cup S}(T)$ and $T_2 = \pi_{B \cup S}(T)$ results in discovery of all frequent patterns in T.*

Proof. Consider a frequent pattern $P \subseteq V$. Since it cannot have items in both A and B (by Lemma 2), P is a subset of either $A \cup S$ or $B \cup S$. For every $P \in \mathcal{F}(T, \epsilon)$, the following are true:

1. If $P \subseteq A \cup S$, then $P \in \mathcal{F}(T_1, \epsilon)$,
2. If $P \subseteq B \cup S$, then $P \in \mathcal{F}(T_2, \epsilon)$.

Therefore, every frequent pattern is discovered in one database at least. Observe that if $P \subseteq S$, then P is discovered in both projected databases. □



b	c	e	f	g
×	×		×	×
	×			×
×				
×		×		
				×
		×		
×	×	×	×	
×	×		×	
×	×			
×	×	×		×
		×		×
		×		
×		×		
		×		×

a	d	e	g	h
			×	
			×	
				×
		×		
×	×		×	×
	×	×		
		×		
×				
	×			×
		×	×	
×	×	×	×	
	×			×
×	×	×		×
	×			
	×			

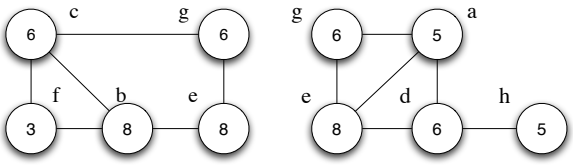


Figure 3.2: *Top:* A GPVS of the G_{F_2} graph of Fig. 3.1. Parts A, B, and separator S are shown. *Middle:* Distribution $D(T) = (T_1, T_2)$ of transaction database. *Bottom:* The G_{F_2} graphs of T_1 and T_2.

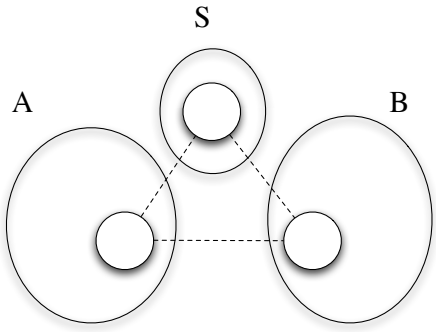


Figure 3.3: Proof by contradiction: assume there were a frequent itemset with a vertex in A, a vertex in B and a vertex in S of $\Pi_{VS} = \{A, B : S\}$ of G_{F_2}. This is impossible since in the GPVS, there cannot be any edges between A and B. Hence, there can be no such frequent itemsets.

Theorem 1 can be improved slightly to suggest a more efficient parallelization. The frequent itemsets within S do not have to be mined redundantly.

Corollary 2 (Collective Work)**.** *Consider the partition of items in Theorem 1. All patterns in T can be mined by mining frequent itemsets that fall within S, independently mining A and B, and then extending frequent itemsets within S by those itemsets mined within A and B.*

Proof. Consider a frequent pattern $P \subseteq V$. By Theorem 1 it is either a subset of $A \cup S$ or $B \cup S$. Assume $P \subseteq A \cup S$. If $P \subseteq S$, then it can be discovered by mining items in S. If $P \subseteq A$, then it can be discovered by mining items in A. Otherwise, we can mine it by merging the frequent itemsets in A and in S using a suitable efficient algorithm (without re-mining frequent itemsets already found in respective item-sets), i.e., extending frequent itemsets in S by items in A until all frequent items are discovered. Likewise for the case that $P \subseteq B \cup S$. Therefore, the mining task has been decomposed into 5 sub-tasks. First, A, B, and S can be mined independently of each other. And then using the results of these sub-tasks, the remaining frequent itemsets in $A \cup S$ and $B \cup S$ may be mined again independently. □

3.1.3 Minimizing data replication

Data replication in distribution $D(T) = \{T_1, T_2\}$ is determined by the vertex separator S. By definition of the two-way distribution, for every transaction $X \in T$, $X \cap S$ is projected in both T_1 and T_2.

Lemma 3. *The amount of data replication in two-way transaction database distribution $D(T)$ given in Definition 4 is equal to $\sum_{u \in S} f(u)$.*

Proof. The frequency function $f(\cdot)$ gives us how many times a given item occurs in T. Since $f(u)$ gives us the size of the tidlist of an item, $\sum_{u \in X} f(u)$ measures the size of data in $\Pi_X(T)$. Since S exists in both $A \cup S$ and $B \cup S$, the amount of data replication is the size of data projected over S. □

The amount of data replication is related to the sparsity of G_{F_2} graph. We expect the replication to grow rapidly beyond a certain edge density that is determined by the support threshold.

Lemma 4 (Minimum Replication)**.** *GPVS of G_{F_2} with item frequencies as vertex weights minimizes the amount of data replication.*

Proof. GPVS of a vertex-weighted graph will minimize the weight of the separator as its partitioning objective. GPVS of G_{F_2} minimizes data replication since the weight of the separator is equal to the amount of data replication. □

Minimizing data replication is also correlated to minimizing the total volume of communication during database redistribution. If the database is to be provided from a central server, then both objective functions are identical. Moreover, for an initial random distribution of the database, we are minimizing the upper bound of total communication volume during the redistribution phase. Note that the GPVS model will maintain storage balance among processors due to the partitioning constraint.

3.1.4 Minimizing collective work

Here we take a look at possible choices for $w(\cdot)$ to minimize collective work. If the computational work estimate for a projection over a set of items X is in the form of a summation of individual load estimates $l(\cdot)$ for items:

$$w(\pi_X(T)) = \sum_{u \in X} l(u) \tag{3.5}$$

then the proposed GPVS model will minimize collective work instead of minimizing data replication. It will also balance computational load due to the partitioning constraint.

Estimating the computational load is non-trivial, since we cannot know in advance how many patterns are present in the data. However, we can reason about the potential number of itemsets in the search space that the mining algorithm will need to traverse. Although every algorithm follows a different strategy for determining frequent patterns, a measure of the portion of the search space containing potentially frequent patterns gives us a good estimate as in [4, 5]. In our method however, computing the maximal cliques in G_{F_2} (like in [5]) will incur additional overhead. Therefore we use simpler functions for load estimation such as the following:

$$w_1(\pi_X(T)) = \textstyle\sum_{u \in X} f(u) \tag{3.6}$$

$$w_2(\pi_X(T)) = \textstyle\sum_{u \in X} \binom{d(u)}{2} \tag{3.7}$$

$$w_3(\pi_X(T)) = \tfrac{1}{2} \textstyle\sum_{(u,v) \in X^2} f(\{u, v\}) \tag{3.8}$$

For estimating computation time, we can use Equation 3.6 which calculates the data size within the projection over a given itemset X in a fashion resembling [4]. Equation 3.6 does not take into account the actual complexity of the task. An alternative approximation, which is inexpensive, can be found in [5]. Equation 3.7 is based on Zaki et al.'s itemset clustering [5] where $d(u)$ is the degree of vertex u in G_{F_2}. This estimate is an upper bound on the number of potential frequent patterns of length 3 obtained by calculating the number of 2-combinations of patterns with length 2. Naturally, more advanced load estimate methods can be used to improve the accuracy. An obvious choice among the simpler functions is

the total frequency of G_{F_2} edges that fall within a given itemset X which gives us Equation 3.8. Although $w_3(\cdot)$ does not strictly conform to Equation 3.5, it can be made so by evenly distributing the weight of each edge among its incident vertices, which yields an approximation to Equation 3.8. In our experiments, we have found that $w_1(\cdot)$ performed better or as well as $w_2(\cdot)$ and $w_3(\cdot)$ perhaps because it tends to reduce both data and task overhead.

3.1.5 Extension to n-way distribution and any level k of mining

We will now show means to extend two-way transaction database distribution to an n-way distribution $D(T) = \{T_1, T_2, \ldots, T_n\}$, where the independent mining conditions are generalized in the obvious way. The two-way transaction database distribution can be applied recursively to divide the two projected databases. Since the resulting projected databases are transaction databases themselves, we can apply the same method to divide them further.

In order to distribute the derived databases, one must obtain the G_{F_2} of the two parts. This can be accomplished by simply running the same algorithm for the projected transaction database, however this can be costly. In the following, we present facts that lead to an efficient computational scheme to calculate an n-way distribution directly over G_{F_2}. By making use of this simple observation, we avoid constructing intermediate projected databases. There is no need to recompute F and G_{F_2}, since they are already known as shown by the following lemma.

Lemma 5 (G_{F_2} of a projection)**.** *For a given itemset $X \subseteq I$, $G_{F_2}(\pi_X(T), \epsilon)$ is the subgraph of $G_{F_2}(T, \epsilon)$ induced by the vertex set X.*

Proof. By the definition of $\pi_X(T)$ and $\mathcal{F}$. □

We thus observe that we do not need to construct intermediate databases to calculate the G_{F_2}'s of the sub-databases in $D(T)$.

Corollary 3 (Fast Recursive Distribution)**.** *Regarding the distribution* $D(T) = \{\pi_{A\cup S}(T), \pi_{B\cup S}(T)\}$ *induced by* $\Pi_{VS}(G_{F_2}) = \{A, B : S\}$*, the* $G_{F_2}(\pi_{A\cup S}(T), \epsilon)$ *and* $G_{F_2}(\pi_{B\cup S}(T), \epsilon)$ *can be calculated as vertex induced subgraphs of* $G_{F_2}(T, \epsilon)$ *by vertex sets* $A \cup S$ *and* $B \cup S$*, respectively.*

The simplest way to obtain an n-way distribution is to use an n-way GPVS directly. Independent mining results extend to the n-way case in an obvious fashion. Thus, we will not prove them separately. However, there are a few differences from the two-way case, which we will now portray. In an n-way GPVS $\Pi_{VS}(G_{F_2}) = \{V_1, V_2, \ldots, V_n : S\}$ of the G_{F_2} graph, we note that the projection of $S \cup V_i$ will result in independent mining. Although S is a minimal separator (i.e., no subset of it is a separator), we observe that not all S need to be replicated in all parts. In general, a portion of S will have to be replicated on processor i (i.e., $Adj(V_i) \cap S$) which may in the worst case correspond to S. This implies that an item in S may be replicated in a different number of projected databases than others in the resulting distribution. The n-way GPVS model does not encapsulate this fact. However, as will be seen, it is easier to implement with an n-way GPVS tool.

Our formulation is also applicable to levels higher than 2 in case G_{F_2} is too dense. We define a graph G_{F_k} of k-length frequent itemsets as follows:

Definition 5. $G_{F_k}(T, \epsilon) = (F, E)$ *is an undirected graph in which each vertex* $u \in F$ *is a frequent item. For each frequent itemset* X *of length* k *in* F_k*, we insert a clique of items in* X *into this graph, i.e., one edge for each length* 2 *support of* X*.*

This definition allows us to use all the relevant results with no modification. The extension of results is trivial and will not be detailed due to space considerations. However, one property is important:

Lemma 6 (Sparsity of Higher Levels)**.** $G_{F_{k+1}}$ *is not denser than* G_{F_k}*.*

Proof. The edge set of $G_{F_{k+1}}$ is a subset of the edge set of G_{F_k} because some of

the k-length frequent items will not be subsets of $(k+1)$-length frequent patterns, hence they will be pruned when constructing the $G_{F_{k+1}}$. □

3.1.6 Maximal and Closed FIM problems

Our method is applicable to both variations of the FIM problem that compute subsets of $\mathcal{F}$. In maximal FIM, no set that is a subset of a frequent itemset is output [3, 84]. In closed FIM, no set that is a subset of a frequent itemset and is supported by the same transactions is output [2]. For instance, consider frequent itemset $X = \{a, b, c\}$. In maximal FIM, no subset of X like $\{b, c\}$ will be output, and in closed FIM, $\{b, c\}$ will be output if and only if it occurs in a different set of transactions than X. After item distribution, if a processor has a set of frequent items X, it also has all transactions belonging to all subsets of X. Thus, both maximal and closed itemset mining can be parallelized with our method.

3.2 Two Data-Parallel Algorithms

In this section, we present *NoClique* and *NoClique2*, which are coarse-grain data-parallel algorithms based on the theoretical observations of Section 3.1. Our algorithms compute the set of frequent itemsets and their frequencies for a given global transaction database T and a support threshold ϵ on n processors. The implementation of *NoClique2* is built upon our new vertical serial FIM algorithm *Bitdrill*.

3.2.1 NoClique: the black-box parallelization

NoClique is a direct application of Theorem 1 and Corollary 3. First, we compute G_{F_2}. Then, we recursively apply the two-way database distribution of Definition 4 until we have n parts, using fast recursive distribution (Corollary 3). For instance, assume $n = 4$. Consider the two-level partitioning that results

in the G_{F_2} graphs of T_1 and T_2 in Fig. 3.2. We have parts A, B, and separator S at the top level; we take two vertex-induced subgraphs of G_{F_2} over $A \cup S$ and $B \cup S$. If we apply GPVS recursively on $G_{F_2}(T_1)$ and $G_{F_2}(T_2)$, we can obtain four overlapping itemsets that define an item distribution such as $D(I) = \{\{b, c, f, g\}, \{b, e, g\}, \{a, d, e, g\}, \{d, e, h\}\}$. Now, each itemset in $D(I)$ can be assigned to a processor. The database is redistributed to processors according to this assignment. Afterwards, we can run any given sequential FIM algorithm on each processor simultaneously and independently, with no further communication. The main advantage of this parallelization is that any serial FIM algorithm that starts from level 3 can be used. The disadvantage is that, since some subgraphs of $G_{F_2}(T)$ are replicated, there is some redundant work. Therefore, this algorithm is suitable only for sparse problem instances that do not require much replication. The recursive application of the two-way item distribution can be carried out in parallel, and of course it is much better if a parallel GPVS algorithm can be used. We have obtained extremely high superlinear speedups in the parallelization of *FP-Growth* and *AIM2* which prompted us to continue research in this direction. We applied *NoClique* to parallelize *kDCI* [11, 85, 29], *LCM* [86] (all FIM), *DCI-Closed* [11], *AIM* [87] (version 2), and *FP-Growth-Tiny* [88].

3.2.2 Bitdrill: our sequential mining algorithm

Bitdrill is a new efficient sequential FIM code that we developed as a basis for our *NoClique2* algorithm. It uses tries (prefix trees) to store sets of itemsets, where each itemset is a string of items in decreasing order of frequency. It uses tidlists (a tidlist is a list of transaction id's an item occurs in) to store the database in memory; linked lists of items are used for sparse items and bit vectors are used for dense items. The algorithm proceeds in BFS order and affords fast candidate generation in a fashion similar to *kDCI* (which is one of the most efficient FIM algorithms together with *LCM*). We use a regular tree data structure instead of prefix arrays in *kDCI*. Fast candidate generation relies on the fact that the prefix tree already captures much of the proximity between two itemsets needed for generating a candidate. Let A and B be two frequent itemsets of length k that

share a prefix of length $k-1$. Both will be the children of the same internal node in the prefix tree. Thus, one can simply take their union and generate a $(k+1)$-length candidate itemset. When we consider the Downward Closure lemma, we will see that all candidates can be generated in this fashion since any subset of a candidate must be frequent and will have frequent subsets with all possible $(k-1)$-length prefixes. Thus, we can simply traverse the prefix tree and generate all candidates by taking 2-combinations of the children of each internal tree node that corresponds to a $(k-1)$-length prefix. After the candidate is generated, it is subject to further pruning employing the Downward Closure lemma. Since we use a vertical representation, the frequency of candidates can be calculated on the fly. To speed up the tidlist intersections, we use a cache to hold all the tidlist intersections in the path to the root, so that a single additional intersection is sufficient to count the transactions in a candidate itemset. The overall algorithm is quite efficient; its performance is comparable to *kDCI* for dense databases and is faster than *kDCI* for sparse databases (due to the dynamic tidlist representation).

3.2.3 NoClique2 algorithm

3.2.3.1 Assumptions

We assume that the number of items is much greater than n (the number of processors). We assume that the database has already been mined up to level l and a GPVS of G_{F_l} has been computed. In the following, we use k as a variable level and we start mining from level $l+1$. Our algorithm will work better when G_{F_l} can be partitioned well. In many cases, there is a suitable l.

3.2.3.2 Overview

Using our n-way GPVS-based item distribution/replication scheme, we decompose the mining problem into a collective work phase (with communication), and independent work phase (with no communication) following the observations in Corollary 2. The algorithm takes as input at each processor a local transaction

database T_{local}, and an absolute support threshold ϵ. We assume that T has been partitioned transaction-wise into T_{local}'s prior to the execution of the mining algorithm. We also supply the set of frequent itemsets up to and including level l, the graph G_{F_l} corresponding to level l, and a heuristic GPVS solution $\Pi_{VS}(G_{F_l})$. The algorithm is comprised of four phases:

1. Redistribute items with selective replication.
2. Mine replicated items in parallel.
3. Mine non-replicated items independently.
4. Merge frequent itemsets across replicated and non-replicated sets of items.

The phases of our algorithm explained in the following.

3.2.3.3 Redistribution of items

Items are distributed according to an n-way GPVS of G_{F_l}. The items in the separator V_s are replicated on each processor. Every other part V_i in the partition contains items collected on a distinct processor. Using the notation of *NoClique*: $D(I) = \{V_i \cup V_s | V_i \in \Pi_{VS}(G_{F_l})\}$.

The horizontal input databases are scanned and using all-to-all personalized communication, each processor receives the parts of transactions that it requires according to the item distribution. After that each processor constructs tidlists of those items.

3.2.3.4 Mining replicated items in parallel

Since each processor has the tidlists of all the items in V_s, we can parallelize candidate generation and testing steps fairly well, starting from level $l + 1$. Assume that for a previous level k, we have the frequent itemsets inserted in decreasing frequency order into a prefix tree. On the prefix tree, we can efficiently

generate candidates for level $k+1$ using fast candidate generation of *Bitdrill*. While traversing an internal node for a $k-1$ length prefix during fast candidate generation (Section 3.2.2), for a children (all of which are leaves) at most a^2 candidates can be generated. Those internal nodes are each given the just mentioned upper-bound of a^2 as weight and we partition the prefix tree into n sub-trees of alphanumerically consecutive itemsets, where each sub-tree has a roughly equal sum of weights. Each processor generates a distinct set of candidates with fast candidate generation on the assigned sub-tree, and then intersects tidlists to check their frequencies, simultaneously. At the end of the iteration, the (locally output) frequent itemsets of length k+1 are gathered on all processors. The iteration continues until frequent itemsets are exhausted. Since both candidate generation and testing steps are parallel, and the sub-tree based distribution of candidates makes local tidlist caches useful, this phase works fairly fast.

3.2.3.5 Independent mining

On each processor i, there is a distinct set of tidlists corresponding to items in V_i not present on any other processor. The frequent itemsets within V_i are mined using a level-wise vertical mining algorithm (*Bitdrill*) starting from level $l+1$.

3.2.3.6 Merging frequent itemsets

As the last step, we mine frequent itemsets that have items in both the replicated V_s and the non-replicated items V_i (for a processor i). We use the output of two preceding phases to achieve this. We start with level $l+1$ again. For merging frequent itemsets in a level $k+1$, assume that we have the frequent itemsets in level k. We use both the frequent items in level k and the already mined frequent itemsets in V_s and V_i to prune as many frequent itemsets as possible. We apply the well-known Downward Closure pruning. Furthermore, any generated candidate must be combined from already-mined two sets of frequent items that we are merging. We have adapted fast candidate generation to work with our itemset merging logic. We have achieved this in two complementary steps explained

below:

First step: For any candidate itemset that has at least 2 items in either part (V_s or V_i), we can use ordinary fast candidate generation over the frequent itemsets in level k that have items in both V_s and V_i sets. After that, we check for a candidate C if $C \cap V_s$ is frequent in replicated database, which is the output of phase (ii), and $C \cap V_i$ is frequent in independent database, which is the output of phase (iii).

Second step: Consider a $(k+1)$-length candidate C with one item x in one part and k items in the other part. Not all of its k-length supports have at least 1 item in either part, therefore C cannot always be generated from the frequent itemsets between parts in level k with fast candidate generation. We make use of the observation that if C is frequent, V_s will have k k-length subsets that include x. While traversing the $(k-1)$ length prefixes in the prefix tree, for each item x, we construct a set of conditional $(k-1)$-length patterns that have items in only one part by removing x. Then, for each item x, we generate k-length candidates from the corresponding set of conditional patterns using fast candidate generation. These k-length candidates have items in only one part and are checked if they are already frequent in that part. If so, then we add x back to generate C and apply Downward Closure pruning restricted to k-length subsets across both parts.

After fast candidate generation, we use the ordinary caching and intersection routines of *Bitdrill* to calculate frequencies. Iteration continues until exhaustion of merged patterns. Note that this step can be used for any distribution of items, not just for our GPVS-based distribution.

3.2.4 Repl-Bitdrill algorithm

The phase of mining replicated items in parallel can be considered as a stand-alone parallel FIM algorithm, which is similar to the second phase of *ParDCI* [10]. When used on its own, we call it *Repl-Bitdrill* as it replicates the tidlists of *all*

frequent itemsets on *all* processors, at the level that it starts mining. Note that *NoClique2* degenerates to *Repl-Bitdrill* when partitioning is impossible, i.e., all items are replicated. *Repl-Bitdrill* is used in Section 3.3 to experimentally show the merits of partitioning in *NoClique2*.

3.2.5 Comparison with Par-Eclat

To put things in perspective, it may help to note the ancestry of our algorithm. Our algorithm is close to *Par-Eclat* [5]. The most important similarities between two algorithms are: (i) We use the same graph of two-items when $l = 2$. (ii) We also use graph theoretic observations to cluster items. (iii) We also distribute items so that each processor mines independently with no further communication. On the other hand, we highlight the following differences: (i) We propose a novel itemset clustering method based on GPVS. (ii) Our item distribution method can minimize data replication by setting vertex weights appropriately. (iii) Our algorithm is fairly independent of the underlying serial mining algorithm (but needs further work to implement phases (ii) and (iv) of *NoClique2*). (iv) Work over replicated items is parallelized.

3.2.6 Implementation

Our implementations of *Bitdrill*, *Repl-Bitdrill*, *NoClique* and *NoClique2* are written in C++ using MPI. The computation of GPVS in *NoClique2* is relevant to the experiments. We use the hypergraph partitioning-based formulation for computing a GPVS of G_{F_l} [89, 90]. To that end, we use the hypergraph partitioner *PaToH* [91, 92].

Database	4 processors			8 processors		
	NC2	RBD	PDCI	NC2	RBD	PDCI
T60.I10.2000K	3.4	2.3	1.2	4.3	4.0	2.3
user-likesmovies	2.9	2.9	2.8	5.0	5.0	7.4
trec	2.6	2.7	–	4.2	4.5	–
trec.lp.200000	2.8	2.8	0.7	4.9	4.9	1.5

Database	16 processors			32 processors		
	NC2	RBD	PDCI	NC2	RBD	PDCI
T60.I10.2000K	7.7	6.2	4.4	11.1	2.6	8.7
user-likesmovies	7.5	7.9	11.4	10.7	10.6	8.2
trec	6.3	5.1	–	8.7	7.1	–
trec.lp.200000	7.9	1.6	2.8	13.0	1.7	5.2

Table 3.1: Speedup Values

3.2.7 Applicability to dense data

We have indicated that our algorithm is not supposed to work well with problem instances that give rise to a dense graph. In dense databases, this is not necessarily the case and we have observed that our method works even with such databases. When the graph is quite dense, a large number of items is replicated and our algorithm degenerates into an algorithm like the second phase of *kDCI* that replicates all items. Often, choosing a more suitable support threshold or a starting level for our algorithm helps.

3.3 Experiments

We have run our algorithms *NoClique2* and *Repl-Bitdrill* as well as *ParDCI* on one synthetic (T60.I10.2000K) and three real-world databases on a Beowulf cluster. In Table 3.1, *NoClique2*, *Repl-Bitdrill* and *ParDCI* are abbreviated as NC2, RBD, and PDCI, respectively. As seen in Table 3.1, out of 16 parallel mining cases, *NoClique2* achieves considerably higher speedup in 8 cases, whereas *NoClique2* and *Repl-Bitdrill* attain close speedups in 8 cases. *ParDCI* achieves the highest speedup in 2 cases. For the trec database, *ParDCI* unfortunately crashed

and we could not measure its running time. We would expect it to have good performance as in user-likesmovies which is similarly dense. For the sparser database T60.I10.2000K, *NoClique2* achieves better speedups than the other algorithms. Only *NoClique2* attains increasing speedup with increasing number of processors for all the databases. *Repl-Bitdrill* and *ParDCI* show this nice property only for 2 databases each.

We now present a detailed explanation of the databases, experimental setup, speedup, partitioning quality, running time dissection, speedups of *NoClique* parallelizations and discussion of observed superlinear speedups in *NoClique*. With regards to partitioning quality, we have examined expected vs. actual load imbalance and data replication ratio. We have seen that our heuristic load estimates work but could be much improved. Data replication is controlled well enough but it is better for small number of processors. It turns out that in the sparse database, independent mining phase dominates and in the dense databases (user-likesmovies and trec) the collective work phase dominates. For these databases the replication approach of *Repl-Bitdrill* and *ParDCI* is effective. However, in an important other case (trec.lp.200000) which represents the "long tail" in a real-world dataset, there is a mixture of both phases and ultimately *NoClique2* does much better than *Repl-Bitdrill* and *ParDCI*, showing the true potential of our approach. The trec.lp.200000 database contains items in the trec database with a frequency of 200000 and lower. In the trec database, it is not possible to mine frequent itemsets beyond a narrow set of items due to the power-law like distribution of items, however in such real-world databases we are interested in relationships among a large number of items.

3.3.1 Data

Table 3.2 shows the synthetic and real-world databases we have used for our experiments. In the table, $|T|$, $|I|$, $|X|_{avg}$ denote the number of transactions, the number of items, the average transaction size, respectively, and the last column gives the database size.

Database	$\|T\|$	$\|I\|$	$\|X\|_{avg}$	Size (MB)
T60.I10.2000K	2×10^6	6000	60	553
user-likesmovies	4.78×10^5	17700	118.6	292.9
trec	1.68×10^6	5267657	177.2	1414.8
trec.lp.200000	1.68×10^6	5267657	138.63	1172.3

Table 3.2: Databases

3.3.1.1 Synthetic database

We have used the association rule generator described in [1] to generate the synthetic database T60.I10.2000K, based on parameters found in previous work such as [5]. The average length of maximal patterns is 10, and the number of patterns is 10000 for the synthetic database.

3.3.1.2 Real-world databases

The user-likesmovies database is derived from the NETFLIX competition, where items are movies and transactions are the movies that a user has liked by rating them more than 3 over 5. The frequent itemsets are movies that many users like together, which can be used for movie recommendations. The trec database is the WebDocs database [93], which is basically a binary term-document matrix derived from the TREC data. The trec.lp.200000 database is constructed in the the same way as WebDocs database but it only includes items with a frequency of 200000 and lower. This database is included because we cannot reach many interesting patterns in a feasible time if we do not discard the few high-frequency items. We cannot use the small (usually smaller than 40 MB) dense databases that are popular in dense data mining research since those have very few items while our algorithm requires many, and obviously replicating a small dense database and mining it with a very low support threshold is best achieved with *ParDCI* [10], *Repl-Bitdrill*, or a similar parallel algorithm, not *NoClique2*. On the other hand, we expect there to be many databases that are not too dense for any decomposition to work, like the real-world databases that we use (in

Database	Support	$\|F\|$	$\|\mathcal{F}\|$	Bitdrill runtime
T60.I10.2000K	5000	4470	2075540	1114.2 sec.
user-likesmovies	20000	716	325806	223.1 sec.
trec	150000	313	587969	329.5 sec.
trec.lp.200000	20000	2225	412057	829.3 sec.

Table 3.3: Problem instances

particular trec.lp.200000).

Table 3.3 depicts the problem instances we have used in the experiments. The second column is the support threshold, $|F|$ is the number of frequent items, $|\mathcal{F}|$ is the number of frequent itemsets, and the last column gives the running time of *Bitdrill* after level 3.

3.3.2 Experimental setup

We measured the performance of our parallel FIM program on a Beowulf supercomputer [94] comprised of 32 compute nodes, a switched Gigabit Ethernet interconnection network and an interface node. Each node has a 3 GHz Pentium IV processor and a local disk, running Linux kernel 2.6. The interface node and the first 16 processors have 2 GB memory each and the rest have 1 GB memory each.

The proposed algorithms *NoClique2*, *Repl-Bitdrill* and the previous algorithm *ParDCI* [10] were run on $4, 8, 16, 32$ processors. Support thresholds are given as absolute. We give the times for *Repl-Bitdrill* because comparing *NoClique2* to *Repl-Bitdrill* shows how much partitioning helps in *NoClique2*. We have acquired the state-of-the-art *ParDCI* [10, 29] and compared it to our algorithms. We have not been able to find any other efficient MPI-based FIM program for comparison. *NoClique2* was run on G_{F_3}, starting mining from level 4. Likewise, *Repl-Bitdrill* and *ParDCI* start mining from level 4, the times for *Repl-Bitdrill* and *ParDCI* include database replication. *ParDCI* decides dynamically when to switch to

a replicated vertical representation. The support threshold that we used for T60.I10.2000K was similar to support thresholds used in previous FIM papers that dealt with sparse databases. We took the 0.25% value from [5].

Sequential `PaToH` was used for HP-based GPVS. We have incorporated the parallel running time for GPVS calculation assuming that parallel PaToH would have 80% efficiency. Since the `PaToH` tool [91] used for GPVS involves randomized algorithms, it yields different transaction database distributions for the same problem instance (graph/hypergraph partitioning programs use randomized heuristics to ensure stable partitioning quality). Thus, each experiment has been repeated 5 times and the average values have been plotted in the following figures.

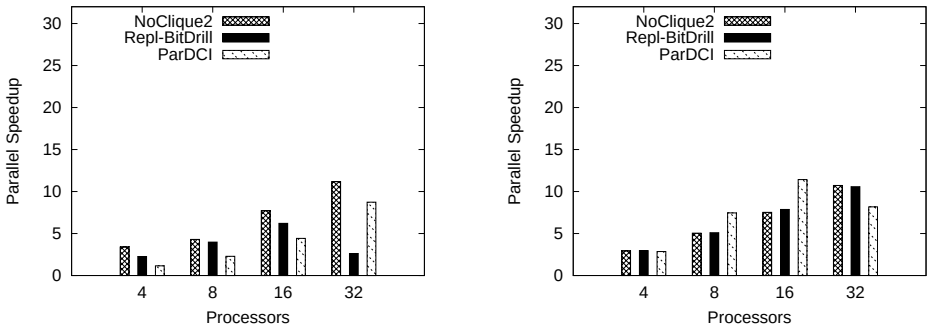
3.3.3 Speedup

Speedup is defined as the ratio of the serial runtime of the best sequential algorithm to the parallel runtime [83, Chapter 2]. The plots in Fig. 3.4 convey the average speedup results for *NoClique2* and the deterministic speedup results for *Repl-Bitdrill* and *ParDCI* on 4, 8,16, and 32 processors. Since *Bitdrill* is the best or close to the best for all of our problem instances, we calculate speedup relative to *Bitdrill*.

As seen in Fig. 3.4, out of 16 parallel mining cases, *NoClique2* achieves considerably higher speedup in 8 cases, whereas *NoClique2* and *Repl-Bitdrill* attain close speedups in 8 cases. *ParDCI* achieves the highest speedup in 2 cases. *ParDCI* has crashed on the trec database, however in that database we expect that it has close performance to *Repl-Bitdrill* as is the case with user-likesmovies which is a dense database like trec.

For the sparser database T60.I10.2000K, *NoClique2* achieves better speedups than the other algorithms. Only *NoClique2* attains increasing speedup with increasing number of processors for all of the databases. *Repl-Bitdrill* and *ParDCI* show this nice property only for 2 databases each.

The speedups of *NoClique2* in Fig. 3.4 pertain to the phases succeeding the



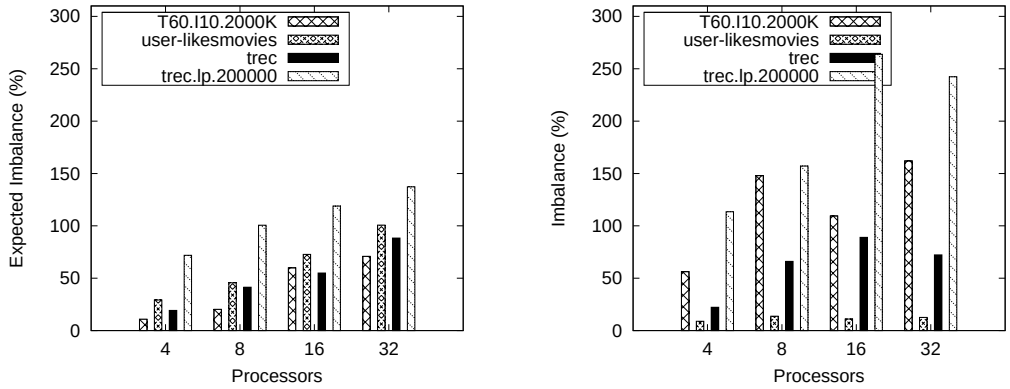
(a) T60.I10.2000K database

(b) user-likesmovies database

(c) trec database

(d) trec.lp.200000 database

Figure 3.4: Speedups of *NoClique2*, *Repl-Bitdrill* and *ParDCI* for the problem instances given in Table 3.3. ParDCI unfortunately crashed on trec database, and those were omitted.



(a) Expected load imbalance

(b) Actual load imbalance

Figure 3.5: Load imbalance of *NoClique2*.

computation of G_{F_3} and its GPVS, including database redistribution overheads. It is seen that the speedups in the portion of the task we are parallelizing are promising.

3.3.4 Partitioning quality

We show the quality of our GPVS model by computational load imbalance and data replication ratio. Figure 3.5 shows the expected and actual load imbalance versus the number of processors for the parallelization of *kDCI*. We define the actual imbalance of a parallel run as $LI_{act} = (t_{max}/t_{avg}) - 1$ where t_{avg} is the average running time of the independent work phase (independent mining and itemset merging together), and t_{max} its maximum. Consider the item distribution $D(I) = \{I_1, I_2, \ldots, I_n\}$, we can define the expected load imbalance by the estimates derived from the distribution as follows:

$$LI_{exp} = \frac{max\{w(\pi_{I_p}(T))|I_p \in D(I)\}}{\frac{1}{n}\sum_{1\leq p\leq n} w(\pi_{I_p}(T))} - 1 \tag{3.9}$$

For the load estimate function, we have used the function w_1 which was also used in implementing the algorithm.

As it would be expected from an estimate that takes into account only data

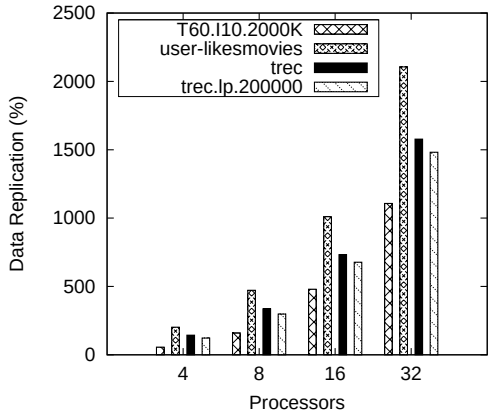


Figure 3.6: Replication ratio of *NoClique2*.

size, the w_1 function does not make precise estimates. However, even the expected imbalances are imperfect, which suggests that improvements in GPVS algorithms could be reflected in our speedups. On the other hand, there is usually a gap between actual imbalance and expected imbalance, suggesting that there is room for improvement in load estimation. For the dense user-likesmovies database, the small actual load imbalance is due to the collective work phase, which dominates the running time for that database.

NoClique2 is geared towards keeping the total amount of data replication low. Figure 3.6 shows the ratio of average data replication for each database and varying number of processors. The replication ratio is measured according to:

$$R_{data} = \frac{1}{|P|} \frac{\sum_{q \in P} \sum_{u \in Iq} f(u)}{\sum_{u \in I} f(u)} - 1, \tag{3.10}$$

where P is the set of processors, I_q is the set of items assigned to processor q. Recall that we are mining itemsets of length $k+1$ and greater.

As seen in Fig. 3.6, data replication increases with growing number of processors as expected. For the denser problem instances, we see that there is more replication. The higher data replication ratio for user-likesmovies database partially accounts for the close performance of *NoClique2* and *Repl-Bitdrill* for this particular database.

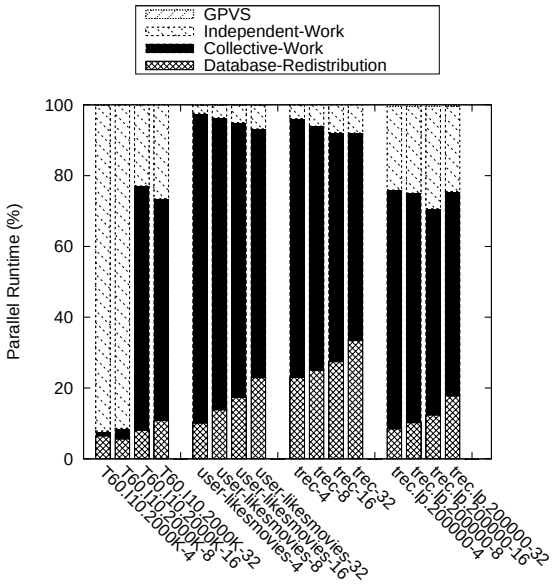


Figure 3.7: Dissection of running time of *NoClique2*.

3.3.5 Running time dissection

Figure 3.7 shows the parallel running time dissection of the three phases in *NoClique2*: database redistribution, collective work (*Repl-Bitdrill* running on the items in separator basically), independent work (comprising independent mining and itemset merging sub-phases), and parallel GPVS. The time dissection is shown varying number of processors. In the labels of the x-axis, the database names and the numbers of processors are separated by dashes, and the y-axis shows the percentage of running time taken by a task. We see that parallel GPVS time is almost insignificant in the running times, it takes about 1% time in trec.lp.200000 experiments.

There are three main sources of parallel overhead in *NoClique2*. First, the more data replication there is, the longer database redistribution takes. Second, the amount of collective work phase can invoke a lot of communication since all the frequent itemsets have to be broadcast at each level. And third, the load imbalance in the independent work phase, comprising independent mining and

merging frequent itemsets, can limit speedup. The influences of those factors depend on the problem instance as our experiments demonstrate.

3.3.6 NoClique parallelizations and superlinear speedups

We have parallelized several sequential (all and closed) FIM algorithms using *NoClique*. However, we will only demonstrate our parallelization of *AIM* and *FP-Growth-Tiny* as they were extremely effective in some cases.

Speedup plots for the black-box parallelization of *AIM* [87] with two synthetic databases are shown in Fig. 3.8. T20.I6.1000K has an average transaction length of 20 and an average maximal pattern length of 6. T40.I8.1000K has an average transaction size of 40, and average maximal pattern length of 8. Both databases have a million transactions. In Fig. 3.8 and the following discussion of superlinearity, speedups are calculated relative to the parallelized sequential algorithm instead of the best sequential algorithm, and they consider the entire mining time. It turns out that our technique speeds up *AIM* significantly. All the speedups in Fig. 3.8 are highly superlinear. For instance, we see a speedup of 63 on 8 processors for 0.002 support. However, note that serial *kDCI* works 20 times faster than serial *AIM* in this case; *kDCI* takes 77 seconds while *AIM* takes 1544 seconds. For that case, AIM is quite slow.

Theoretically, superlinear speedup is not possible except when the serial algorithm that we are comparing to is suboptimal (provided that the serial computer has the same amount of memory). This can be easily shown by noting that the parallel algorithm can be simulated by a serial algorithm. In the significant speedups that we observe, it is indeed possible to speed up the serial algorithm by simulating our parallel program on a serial computer. This will of course be possible only if the serial computer has enough RAM, thus for the larger databases we could not do that. We will mention two cases. First, *NoClique* parallelization of *FP-Growth-Tiny* with the T40.I8.500K database (a smaller version of T40.I8.2000K), using a support threshold of 0.4% run on a single processor simulating 4 processors completes in 31.5 seconds, while the original serial algorithm

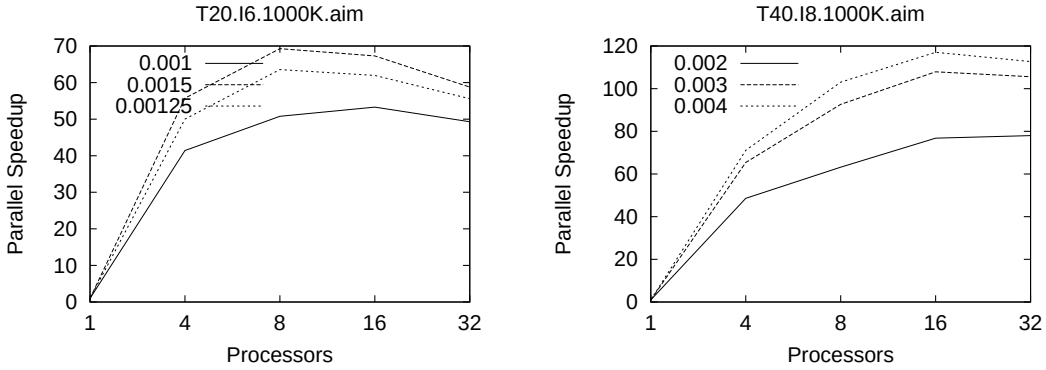


Figure 3.8: Relative speedups for *NoClique* parallelization of AIM on T20.I6.1000K and T40.I8.1000K using various relative supports (1 is 100%).

runs in 353.4 seconds, achieving 11.2 speedup on a single processor. Likewise, *NoClique* parallelization of *AIM* with T40.I8.1000K using 0.4% support on a single processor simulating 4 processors completes in 53 seconds compared to 1146 seconds for the original serial algorithm, which is a speedup of 21.6 on a single processor. Simulation of multiple processors on a single processor was done simply by using a `hosts` file that included a single host in the LAM/MPI implementation.

Therefore, the superlinear speedups basically show that the serial algorithms used were not the fastest for those sparse problem instances. Our interpretation of the superlinear speedup results is that our decomposition method amends scalability issues with some of the serial algorithms. It also makes better use of the aggregate memory of the parallel system and the memory hierarchy of modern processors. Interestingly, we have seen that our method can bring the *NoClique* parallelization of slower algorithms like *FP-Growth-Tiny* to the performance level of the *NoClique* parallelization of a faster algorithm (i.e., *kDCI*).

Since our distribution method tends to reduce the number of items stored per processor, it can be helpful with FIM algorithms as they use costly data structures to represent and process candidate sets. We have found that both *FP-Growth-Tiny* and *AIM* are sensitive to the number of items. In the case of *FP-Growth-Tiny*, the number of items directly influences execution time, since

for each item a new conditional Fp-Tree is constructed (recursively) [15].

After all, we do not really narrow down the search space, we just make it easier for the programs to handle parts of the search space at once. Decomposing the whole search task into a sequence of search sub-tasks can lead to faster mining. Note that all the serial runs that we give do fit into physical RAM and they never thrash the disk.

Chapter 4

Intelligent Candidate Distribution with Selective Item Replication

4.1 Introduction

We extend the Graph Partitioning by Vertex Separator model for parallel FIM we had proposed earlier, by relaxing the condition of independent data mining. Instead of finding independent sets, we may minimize the amount of communication and partition the candidates. When we do not require independent mining in the sense of NoClique2, we may partition the candidates in a fine-grained manner, amending the load imbalance problem that emanates from the coarse-grained load balancing approach of the NoClique2 algorithm. Instead of frequent itemsets, candidates of succeeding levels are generated and we construct the following model which is inspired by the hypergraph transformation of the GPVS problem: each vertex represents a candidate and each hyperedge represents an item. The hyperedges in the cut correspond to items that must be replicated on some processors. Thus, the model minimizes the cost of data redistribution, in similar manner to the chapter on the GPVS model. However, it also finely balances the

parallel load, as each candidate is a task. While this approach has better load balance, it will also have different parallel overheads due to the need to generate many candidates and solve multiple partitioning problems. The model can be calculated only for 4-5 levels and then re-partitioning has to be made until all levels are mined. Otherwise, there would be too many vertices. The partitioning model for this algorithm has been implemented with PaToH and has been seen to work quite well on sample datasets. The re-partitioning model is yet to be worked out; it is an important optimization as it must reduce further data re-distribution volume. An optimization we are considering in addition to re-partitioning is to use a variable threshold for each item, for instance by a ratio of the frequency of each item, so that we discard superfluous frequent patterns and obtain sparser hypergraphs.

Section 4.2 introduces the hypergraph partitioning model. The Intelligent Candidate and Item Distribution algorithm (*ICID*) is introduced in Section 4.3. Section 4.4 proposes a re-partitioning model that augments the hypergraph partitioning model with fixed processor vertices. Section 4.5 explains the implementation details of *ICID*. Finally, Section 4.6 presents our performance study based on the experiments of Section 3.3.

4.2 Hypergraph Partitioning Model

We introduce a hypergraph partitioning model for fine-grain load balancing and item distribution for the parallel frequent itemset mining problem. The hypergraph partitioning model improves substantially over the graph partitioning by vertex separator (GPVS) model (NoClique2 algorithm), as it exerts precise control over both task and data distribution at the same time.

The hypergraph partitioning model is motivated by the observation that the task distribution induced by GPVS does not yield very low load imbalance. The

indirectness and the coarse-graininess of task distribution contribute to the inflated load imbalance in the case of NoClique2 algorithm. To amend this shortcoming, we have revisited the problem with the express purpose of repairing load imbalance. While the GPVS model fares well in terms of decreasing data replication substantially, and finding independently mined subsets merely from supplied frequent itemsets, with more input data, it is possible to make better decisions regarding task distribution in parallel FIM.

Let a hypergraph $H = (V, E)$ where V is a set of vertices, and $E = \{X | X \subseteq V\}$, a set of hyperedges, where each hyperedge X is a proper subset of V. Vertex weights are given by the weighting function $w_V : V \to R$, and hyperedge weights are given by $w_E : E \to R$. The weighting functions are extended to sets with $w_V^s : 2^V \to R$, $w_V^s(X) = \sum_{x \in X} w_V(x)$, and $w_E^s : 2^E \to R$, $w_E^s(A) = \sum_{X \in A} w_E(X)$.

First, we formally define weighted hypergraph partitioning.

Definition 6. (n-way weighted hypergraph partitioning) $\Pi(H) = \{V_1, V_2, ..., V_n\}$ *is a set-theoretic partition of* V *where the weight of the hyperedge-cut* $w_V^s(E_C)$ *is minimized, and where*

$$E_C = \{X \subseteq V |\ \exists V_i, V_j \in \Pi(H) \wedge V_i \neq V_j \wedge X \cap V_i \neq \emptyset \wedge X \cap V_j \neq \emptyset\}, \quad (4.1)$$

under the constraint that for all $V_i \in \Pi(H)$ $w(V_i) \cong w(V)/n$ *[parts have roughly equal vertex weights].*

A hyperedge-cut E_C in this context is a set of hyperedges, removal of which disconnects the hypergraph into separate sub-hypergraphs. Hypergraph partitioning optimizes E_C such that its weight is minimized, and the weights of vertex parts are balanced.

Hypergraph partitioning can model the parallel computation of the FIM problem succeeding level k, for the input parameters transaction database T and support threshold ϵ as follows.

Definition 7. *Let* $H(F_k, l) = (C, \mathcal{N})$ *be a hypergraph where* C *corresponds to the set of candidate itemsets generated from* F_k *up to level* l*, and each hyperedge* X

in the set of nets N corresponds to an item $x \in I$ that occurs in all candidates that are incident on X. Let $i : \mathcal{N} \to I$ store the item that corresponds to a given hyperedge.

The vertices represent atomic tasks, and the hyperedges represent data dependencies in this partitioning model. We assign a vertex to each candidate to establish fine-grain task distribution, while we assign a hyperedge to each item to minimize communication volume.

The weight functions w_V and w_E respectively correspond to load estimate of a candidate itemset, and datum size of an item. Load estimate of a candidate itemset depends somewhat on the serial FIM algorithm used:

1. It may be given constant weight 1, neglecting differences between candidates.

2. Similarly, for efficient vertical algorithms that use a tidlist cache over a prefix-tree such as DCI and Bitdrill, it may be given by

$$w_V(X) = min(\{f(x)|x \in X\}) \tag{4.2}$$

   for candidate set X, since that is an upper bound on the number of items that the tidlist intersection algorithm will perform at the fringe of the prefix-tree, assuming the intersection of other tidlists of X are present in the cache.

3. Conceivably, even more elaborate probabilistic load estimates may be proposed. One such estimate weighs and sums the frequencies of items in the candidate set X. First, the items are sorted in order of decreasing frequency as that is the order they will be inserted in the prefix tree. For the ith item $x = X[i]$ in this order, we add $f(X[i])/(|X| - (i-1))$ to the weight of this candidate itemset. This weight informs that it is much more likely for the tidlist intersections of the more frequent items to be available in the cache, and the least frequent item at the end will incur full scanning cost.

The datum size of an item is its frequency, i.e., length of its tidlist, $w_E(Y) = f(i(Y))$.

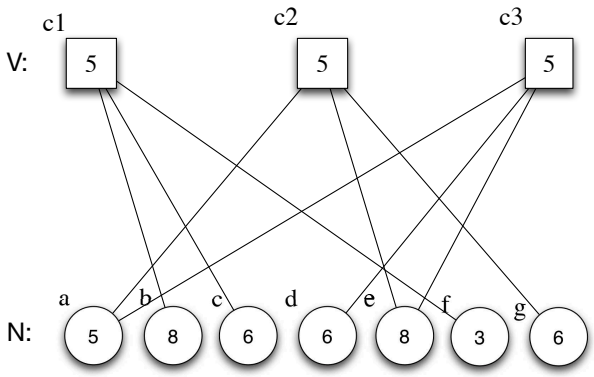


Figure 4.1: Hypergraph model of parallel FIM task for the example database of Fig. 3.1.

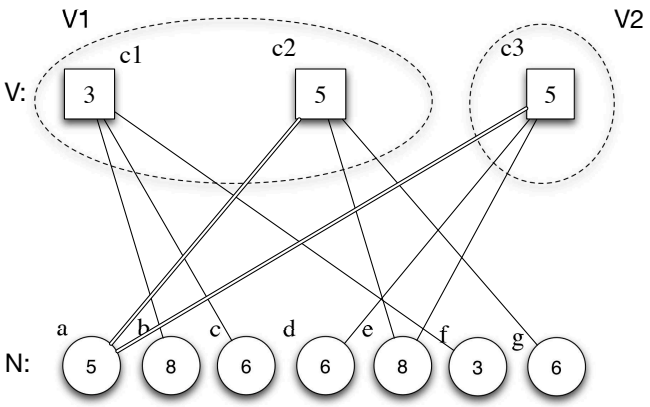


Figure 4.2: A bi-partition of the hypergraph model in Fig. 4.1.

The example of Fig. 3.1 has been adapted to the hypergraph partitioning model in Fig. 4.1. The vertices (candidates) are drawn as squares, and the hyperedges (items) are drawn as circles. The weights are written inside vertices and hyperedges. The labels are attached right next to vertices and hyperedges. The candidate itemsets are labeled as cx. A solution of this partitioning problem is depicted in Fig. 4.2. The dashed lines indicate vertex sets, and the double-lined hyperedge connections indicate hyperedges that belong to the edge-cut.

As in NoClique2, we may assume that the data is initially partitioned transaction-wise randomly, and has to be redistributed according to the partitioning model. Then, the partitioning objective minimizes communication volume under some conditions.

Let us first define the replication size of a hyperedge X with respect to a given partition $\Pi(H)$.

Definition 8. *The replication size of a hyperedge X in $H = (V, N)$ is defined with a size function $\lambda : \mathcal{N} \mapsto \mathtt{N}$ where*

$$\lambda(X) = |\{V_i \mid V_i \in \Pi(H) \wedge X \cap V_i \neq \emptyset\}|. \tag{4.3}$$

λ calculates the number of vertex sets in the partition Π that hyperedge X is connected to.

That is, the item that corresponds to X must be replicated on $\lambda(X)$ processors.

Theorem 2. *Let E_C be the hyperedge-cut corresponding to a weighted hypergraph partition $\Pi(H(F_k, l))$. Each hyperedge X in the cut is incident on some vertex parts, and the item $i(X)$ that corresponds to it must be replicated on those processors. Then, total communication volume is*

$$V = \sum_{X \in \mathcal{N}} \lambda(X).w_E(X) \tag{4.4}$$

for distributing the database T from a central server according to the partition.

Proof. For each hyperedge that is incident on a vertex part, the communication volume increases by the datum-size (number of tid's) of the item that corresponds to the vertex part, which is given by w_V. □

The communication volume for a central server is approximately the same for random horizontal distribution of the transaction database.

The hyperedges that do not remain in the hyperedge-cut, correspond to items that are communicated only once to a home processor. If the cut weight is zero, then the communication volume adds up to $w_E^s(E)$. This lemma illustrates a relation to the $GPVS$ model.

Lemma 7. *Let E_C be the hyperedge-cut corresponding to a weighted hypergraph partition $\Pi(H(F_k, l))$. If we assume that each hyperedge in the cut is incident on all vertex parts, then $V = n.w_E^s(E_C)$.*

Note that this only approximately holds since the hyperedge-cut likely contains large hyperedges, but not necessarily ones that are incident on all vertex parts. Therefore, $\Pi(H(F_k, l))$ relates to minimizing communication volume for redistributing the database T since an initial transaction-wise random distribution is similar to the case with the central server.

To exactly minimize communication volume, we would have to change the partitioning objective to minimize the sum of $\lambda(E).w(E)$ for each edge E in the hyperedge-cut, where $\lambda(E)$ is the replication-size of hyperedge.

We may assume that each candidate itemset may be mined independently, given its data dependencies are present on a processor, which our partitioning model eloquently ensures. Then, each vertex part V_i may be mined on a separate processor $1 \leq i \leq n$.

Lemma 8. *For all vertex parts in the partition $V_i \in \Pi(H(F_k, l))$, let the candidate itemsets in V_i be assigned to processor i. If $w_V(X)$ is a load estimate of candidate itemset X, then $w_V^s(V_i)$ is a cumulative load estimate of processor i.*

Given these general observations, we may design a parallel FIM algorithm that distributes candidate itemsets with selectively replicated item distribution.

4.2.1 Comparison to GPVS model

The primary computational differences of the hypergraph partitioning model are: it represents both tasks and data separately using a hypergraph, it uses different weights for task and data, and it uses the set of candidates to be mined in the levels from $k+1$ up to l instead of the frequent itemsets F_k. These features endow the hypergraph model with much flexibility compared to the graph partitioning model, as now load is balanced in a fine-grained manner. The hypergraph model also performs selective item replication differently, as each item is not always replicated on each processor, and it is only sent to the processors that absolutely need it.

On the other hand, the need to generate candidates in the successive levels to be mined is a slight shortcoming, as this process is likely to generate spurious candidates for large l. Additional parallel overhead will arise from having to redistribute the database multiple times, and therefore compute multiple hypergraph partitionings. We will later show how the consequent partitionings may be optimized.

4.3 Intelligent Candidate and Item Distribution Algorithm

We propose a new parallel algorithm called Intelligent Candidate and Item Distribution (ICID) algorithm which employs the theoretical foundation of Section 4.2 in a practical manner.

We assume that the transaction database $T = \bigcup_{1 \leq i \leq n} T_i$ has been partitioned horizontally in any way prior to the algorithm. The algorithm also takes as

input parameters, the support threshold ϵ, k, l, and naturally the number of processors n. Like NoClique2, we first mine all frequent itemsets, level-wise, up to level k. Any level-wise horizontal parallel FIM algorithm patterned after Count-Distribution may be used for this purpose.

Then, we construct $H(F_k, l) = (C, \mathcal{N})$ and invoke an appropriate weighted hypergraph partitioning tool to approximately solve the optimization problem.

After this step, we know how to distribute items with selective replication, and to distribute candidate itemsets. We assume that each processor has access to the entire hypergraph partition and that the root processor has the entire set of candidates C to be mined. Each processor i is sent the candidate itemsets $C_i \in \Pi(H)$, corresponding to vertices in vertex part V_i.

A collective communication step is performed, and the horizontally distributed transaction database is redistributed such that each processor i receives the tidlists of hyperedges connected to candidate itemsets in V_i. This communication step ensures that after item distribution with selective replication, each processor can proceed independently.

Thereafter, each processor mines its local transaction database that contains a number of tidlists simultaneously and independently, such that mining is restricted to only the candidate itemsets in C_i on processor i.

4.4 Re-partitioning Model for Incremental Algorithm

Since l is usually a small number, multiple iterations of the algorithm are necessary for mining the full set of frequent itemsets $\mathcal{F}$, exactly $q = \lceil (L(\mathcal{F}) - k)/l \rceil$ times where $L(\cdot)$ is the number of levels in a set of itemsets. It seems a practical choice to decrease the number of iterations, to prevent the solution of too many hypergraph partitioning problems. However, if the database is loaded from the horizontal parallel store and distributed from scratch each time, this would result

in excessive parallel overhead.

Instead, we must preserve the already distributed parallel vertical database in memory, or on secondary storage, and then make as little additional communication as possible to process the new set of candidates in the next iteration. We can solve this optimization problem by incorporating the current distribution of items D (Equation 4.9) into our hypergraph partitioning model. This results in a re-partitioning model which distributes the candidates as before, while minimizing the volume of item re-distribution, incrementally. Thus, the correctly construed re-partitioning model that we are about to present becomes a major optimization that we recommend for the *ICID* algorithm.

The modeling of the present distribution of items $D(I)$ is achieved by adding fixed sink vertices corresponding to each processor $p \in 1 \ldots n$. See the one-phase remapping method of [37] for an example of a hypergraph partitioning approach which used fixed vertices. A fixed vertex is defined as follows:

Definition 9. (Fixed Vertex) *A fixed vertex in hypergraph partitioning $\Pi(H)$ is a vertex that is pre-assigned, and cannot be moved. Fixed vertices V_F have been assigned their partition number with a function $\tau : V_F \mapsto 1 \ldots n$.*

We propose the re-partitioning model as follows. Assume that *ICID* is performed in q iterations. Let D_j be the item distribution at iteration j, for all $1 \leq j \leq q$. Let us assume $D_0(x) = \emptyset$ for each $x \in I$. If we assume that we have D_{j-1}, then we can decide D_j with a hypergraph partitioning model. We denote the partitioning at iteration j with Π_j, starting from iteration 1. Let C_j be the candidates generated for iteration j, and N_j be the nets for iteration j in Definition 7. Let $E_{C,j}$ be the edge-cut of the partition at iteration j.

Definition 10. *(Re-partitioning Model) For re-partitioning the candidates in iteration j, Definition 7 is reused where $C = C_j$ and $N = N_j$ augmented with a model of present item distribution D_{j-1}. We add fixed vertices corresponding to each processor, such that for all $p \in 1 \ldots n$, there is a unique vertex $u \in V_F$, and $\tau(u) = p$, where $V_F \cap V = \emptyset$ and $V_F = |n|$. The vertices in V_F are weighed zero since the candidates corresponding to them have already been mined. The distribution D_{j-1} of the items with selective replication is modeled with the addition*

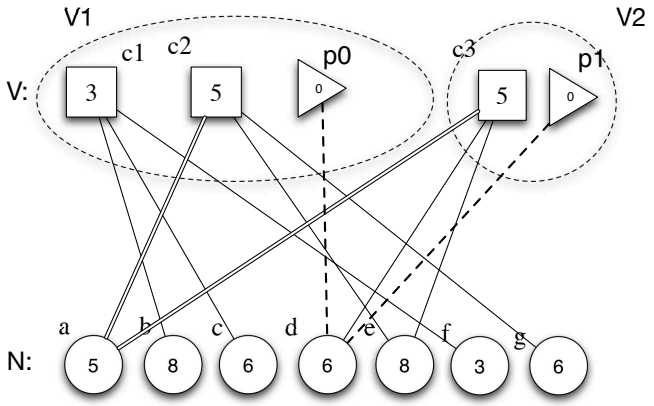


Figure 4.3: Adding fixed vertices to the hypergraph partitioning model of Fig. 4.2.

of the already distributed processors to the new hyperedges that correspond to the items that depend on candidates in C_j. More formally, we update the nets with: $\forall X \in N_j, X \leftarrow X \cup D_{j-1}(i(X))$. After that, the new item distribution D_j is determined from the new partition Π_j.

We show how the fixed vertices are added in Fig. 4.3, where the fixed processor vertices corresponding to each processor are represented with triangles and labeled as $p0$ and $p1$. The processor vertices in the example show that item d has been replicated on both processors prior to re-partitioning (for some reason), and the dashed bold lines show the hyperedges in the edge-cut that are caused by fixed vertices; the partitioning result here does not change, it is intended as a schematic example only.

Definition 11. *Let the re-distribution of the items at iteration j may defined as*

$$R_j[X] = D_j[X] - D_{j-1}[X] \tag{4.5}$$

For mining at iteration j to succeed, only additional replication of items defined by R_j is sufficient.

Let the stand-alone communication volume of iteration j be $V_j = V$ of Equation 4.4 where $N = N_j$.

Definition 12. *The communication volume for re-partitioning at iteration j is*

$$V_j^R = V_j - V_{j-1}. \tag{4.6}$$

This definition is correct, because the distribution according to D_{j-1} has already been achieved in the previous iteration $j-1$. Thus, the volume of communication that has been incurred from iteration $j-1$ is negligible during iteration j. The following theorem clarifies the claim.

Theorem 3. *Re-partitioning model of Definition 10 minimizes communication volume V_j^R.*

Proof. The fixed vertices V_F entail V_{j-1} communication volume, for V_{j-1} is equal to the sum of weights of hyperedges in iteration j that are in the edge-cut E_c among vertices in V_F. Thus, minimizing V_j is sufficient to minimize V_j^R. This is easily proven by observing that the partitioning objective

$$\text{min. } V_j - V_{j-1} \tag{4.7}$$

is equivalent to

$$\text{min. } V_j \tag{4.8}$$

since $V_j \geq V_{j-1}$ and V_{j-1} is constant during iteration j. □

The *ICID* algorithm may be modified in a straightforward manner to obtain the incremental variant. The re-partitioning model of Definition 10 is used in iteration j instead of Definition 7 using the previous item distribution D_{j-1}. Other parameters are maintained and defined as explained above. The database redistribution step assumes that the items in D_{j-1} are already distributed/replicated as such, therefore it merely replicates items in map R_j accordingly.

4.5 Implementation

ICID algorithm is implemented in similar manner to NoClique2.

We have adopted the serial miner Bitdrill, and extended it such that we can now start from a level k, with given frequent itemsets F_k, and restrict all mining to a given set of candidates.

As in NoClique2, we do not particularly focus on the steps prior to partitioning. That part of processing is not relevant to our present algorithm, therefore we do not take it into account in our experiments. In the present implementation, either the serial miner Bitdrill or Repl-Bitdrill may be used for computing F_k. We use a python script to drive the various C++ programs invoked during various stages of the algorithm, this a commonly used implementation method for data mining which combines the flexibility of Python with the efficiency of C++.

We construct the hypergraph model $H(F_k, l)$, which requires the computation of the set C of candidate itemsets from F_k, up to level l. We accomplish this on the root processor and we do not investigate the parallelization of this step. For this reason, we focus on problem instances where the candidate generation is not costly; it is usually much cheaper than mining since it does not use T. We have assigned unit weight to candidate itemsets presently. For hypergraph partitioning, we use PaToH [91, 92], where we balance on vertices – tasks in our model, and minimize the net-cut – hyperedge-cut, by selecting connectivity-1 weighting in PaToH. After this, the present implementation distributes candidates C according to $\Pi(H)$, each $V_i \in \Pi(H)$ corresponds to the set of candidate itemsets that are to be sent to processor i. A distribution map D is calculated as follows, for any item it maps items to processor sets:

$$D[x \in I] = \{i \mid x \text{ is incident on } V_i\}. \tag{4.9}$$

The present implementation partitions C on the root processor and then distributes them to n processors according to the hypergraph partition with a parallel C++/MPI program. Finally, a C++ program takes as input on processor i, the horizontally distributed transaction database T_i, support threshold ϵ, F_k, C_i, $\Pi(H(F_k, l))$, and D which contains the main body of the parallel FIM algorithm. The data is first distributed according to D which provides the information required to distribute and replicate items. We have implemented this step similarly to NoClique2's redistribution where we assume horizontal database representation. Let $M_{n\times n}$ be a message buffer where $M_{i,j}$ is a send buffer from processor i to processor j. Processor i iterates over transactions in its local T_i. For each transaction $X \in T_i$, we can calculate which projection of X must be sent to which

processor in the following manner. For each item $x \in X$, we append x to $M_{i,p}$ for each $p \in D[x]$. Then, we perform an all-to-all personalized communication (AAPC) using the message buffer M. The second phase of the parallel miner uses the aforementioned candidate set restricted bitdrill variant. On completion, the frequent patterns are output to local disk on each processor in parallel, and then the performance measurements are calculated and output.

4.6 Performance Study

We present a performance study to prove that the performance of ICID algorithm is sound. As with NoClique2, we measure both the actual and expected performance parameters, to be able to compare how successful the model has fared in practice. We have measured the following parameters in similar fashion to Section 3.3:

1. Parallel time: the total time, and the time for redistribution and independent mining phases.
2. Preprocessing time: the serial time taken by candidate generation and the PaToH parallel hypergraph partitioning tool.
3. Load imbalance: actual load imbalance, and load imbalance in the hypergraph model
4. Data replication: the amount of data that has been replicated in proportion to the database T

4.6.1 Experimental Setup

We have used the same distributed memory supercomputer in Section 3.3 for a healthy comparison, however, these results were seen to be quite similar on a shared-memory machine, as well, therefore we do not think that the results

Database	Support	l	$\|F\|$	$\|\mathcal{F}\|$	Bitdrill runtime
T60.I10.2000K	5000	6	4470	1367233	645.2 sec.
user-likesmovies	20000	1	716	195130	134.6 sec.
trec	150000	313	2	173603	196.7 sec.
trec.lp.200000	20000	2225	1	76508	1051.0 sec.

Table 4.1: Problem instances

are specific to distributed memory. We have run the experiments on up to 16 processors, as all compute nodes were not available at the time. We have also used the same problem instances, first given in Table 3.3.

The basic difference in the experiments is that we are not completing the whole mining, only as many levels of mining as can be mined succeeding level 3. The running time of these restricted problem instances is summarized in Table 4.6.1. In the table the second column is the support threshold, l is the number of levels mined after level 3, $|F|$ is the number of frequent items, $|\mathcal{F}|$ is the number of frequent itemsets, and the last column gives the running time of *Bitdrill* after level 3 until level $3 + l$. As the table shows, the single synthetic database allows more levels to be mined at once, this is due to its sparser structure.

Our main goal in this experiment is to show that the load imbalance problem of *NoClique2* is addressed by the *ICID* algorithm, and that the new parallel overheads are not prohibitive.

4.6.2 Partitioning quality

We examine whether the partitioning model improves partitioning quality with respect to *NoClique2* in the *ICID* experiments, especially load imbalance. In the experiments, the second weight function, which is the minimum frequency of items in a candidate itemset, was used for vertex weights.

Expected load imbalance, according to the hypergraph partitioning model,

was calculated as follows:

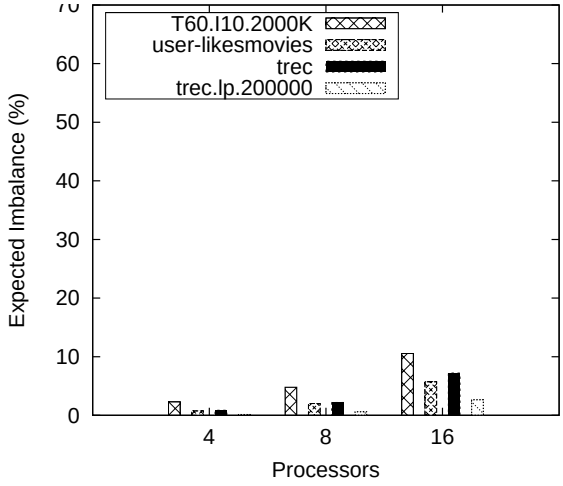
$$LI_{exp} = \frac{max\{w_V^s(V_i)|V_i \in \Pi(H(F_k, l))\}}{w_V^s(V)} - 1 \tag{4.10}$$

Fig. 4.4 shows the expected and actual load imbalance of *ICID* algorithm. We observe that there is a very significant reduction in load imbalance compared to Fig. 3.5. We also observe that the gap between expected and actual load imbalance is fairly narrow, compared to *NoClique2*, suggesting that the finer load estimate modeling works to our advantage. The worst load imbalance is that of the synthetic database, which suggests that it may be due to either its higher sparsity or synthetic origin. There is consistently low imbalance in real-world database, which may be attributed to the numerosity of frequent itemsets, which has likely allowed greater freedom for load balancing.

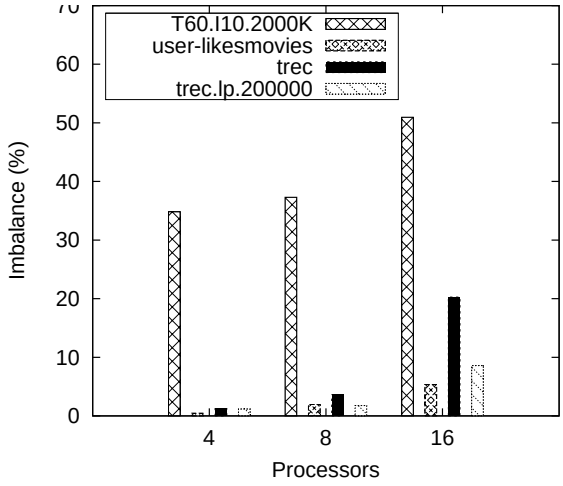
The promising improvement in load imbalance brings us to question whether the partitioning quality suffers in some other respects compared to the GPVS model. We use the data replication ratio given in Equation 3.10. We show the data replication incurred by the *ICID* algorithm in Fig. 4.5, which seems comparable qualitatively to *NoClique2*, although of course we must mention that the full mining case would likely increase data replication a bit. Qualitatively, there is no significant disadvantage in data replication that we my affirm in these results, which shows that the partitioning quality overall in the hypergraph partitioning model has substantially improved without suffering any apparent damage. In particular, the high data replication ratio of user-likesmovies database is brought down compared to Fig. 3.6. The replication ratios of other databases behave similarly to *NoClique2*.

4.6.3 Running time dissection

We analyze the running time dissection of the algorithm for large databases and varying number of processors, which is conveyed in Fig. 4.6. In the labels of the x-axis, the database names and the numbers of processors are separated by dashes.



(a) Expected load imbalance



(b) Actual load imbalance

Figure 4.4: Load imbalance of *ICID*.

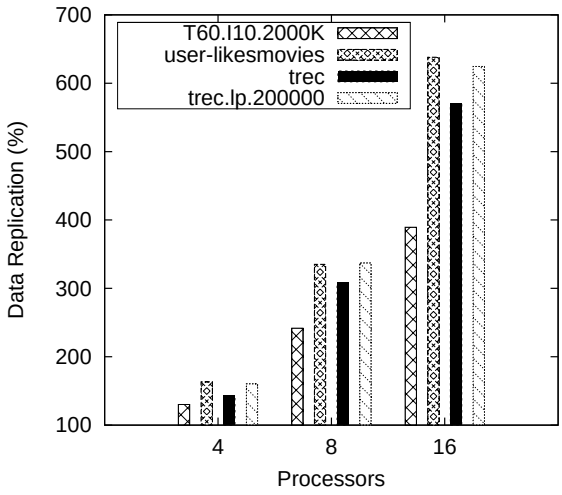


Figure 4.5: Replication ratio of *ICID*.

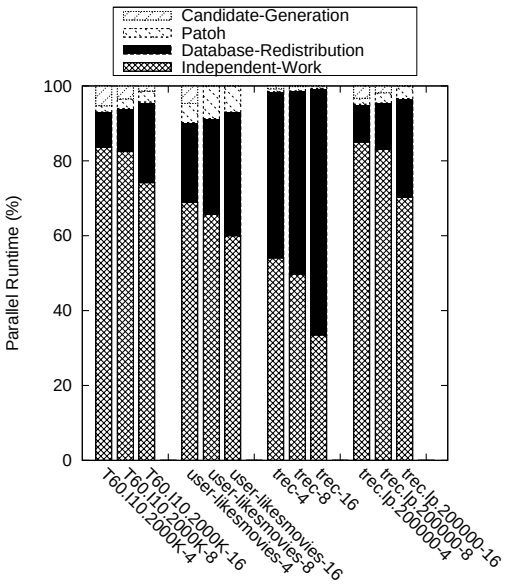


Figure 4.6: Dissection of running time of *ICID*.

The time dissection shows four phases of *ICID* algorithm, which are the candidate generation, hypergraph partitioning (PaToH), Database Redistribution, and Independent Work, in the order of execution. We assume that the preprocessing phases of candidate generation and PaToH may both be parallelized with 80% parallel efficiency and scale linearly, and their contribution is calculated from sequential running time. The dissection shows a similar pattern to Fig. 3.7, and the time is usually dominated by independent work, except for trec database. As in the case with *NoClique2*, database redistribution takes more percentage with increasing number of processors. The database redistribution of the full mining algorithm should take less percentage, since full redistribution will not be necessary for each iteration of the re-partitioning algorithm. Also, interestingly, hypergraph partitioning seems to be as fast, or faster in percentage compared to the independent mining model of *NoClique2*, which is quite encouraging, in terms of signaling that a more complete implementation is likely to show excellent speedup.

There are three parallel overhead differences in *ICID* compared to *NoClique2*. First, the candidate generation step may generate more candidates compared to the candidate generation of plain *Bitdrill*, for when mining multiple levels at once, certain pruning steps cannot be applied to the candidates after the first level of candidates generated. Secondly, the candidate partitioning model may get a bit more expensive compared to *NoClique2*, since there may be many candidates after level 3, which would keep the percentage of the PaToH step constant in time dissection, even for the complete implementation. Thirdly, the more precise load imbalance may sometimes increase communication volume, although we have not seen that in present experiments.

4.6.4 Speedup

Our goal is not to demonstrate total speedup, but to demonstrate that speedup can be obtained even when the whole database is redistributed for mining a small number of levels. Interestingly, we can still obtain significant speedup in this case, showing that the full culmination of our algorithm is likely to result in excellent

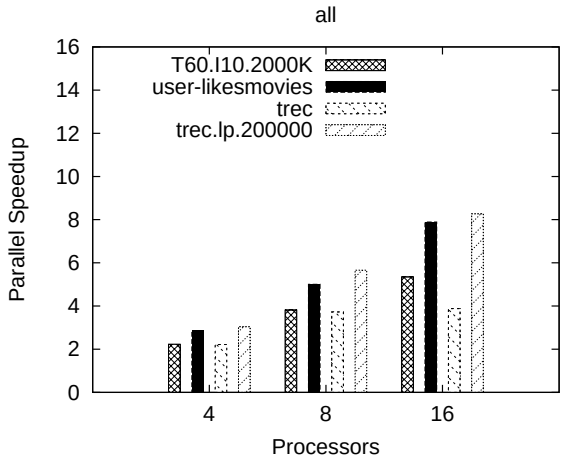


Figure 4.7: Speedup of *ICID* for various databases.

performance. Fig. 4.7 demonstrates the speedup for the problem instances in Table 4.6.1. Note that these speedups suffer the redistribution (with replication) of the entire database, while mining only a part of the tasks in Table 4.6.1. For entire tasks, the speedup should surpass that of *NoClique2*. Despite this major shortcoming, *ICID* still outperforms *NoClique2* for user-likesmovies and trec.lp.200000 databases, proving the effectiveness of our candidate partitioning model beyond any doubt.

4.6.5 Discussion

We use a particularly interesting problem instance from the chapter on NoClique2 algorithm to demonstrate our results in detail. The problem instance is the trec.lp.200000 database with 20000 support threshold, which proved to be the most challenging one for the GPVS model. In particular, the GPVS model has a lot of difficulty balancing this instance, actual imbalance was always more than 1, even for 4 processors. This instance requires 1051 seconds for the fourth level, on a single processor. All the *NoClique2* problem instances were chosen carefully to have several levels after 3. To simplify our understanding, we set k to 3 and

l to 1 for this particular problem instance, as seen in Table 4.6.1 . We present the performance of ICID algorithm on 16 processors. The results turned out to be extremely promising. The run took 136.3 seconds, which corresponds to 8.27 speedup. This is already better than the corresponding *NoClique2* result of 7.9. PaToH obtained near-perfect load balancing, a load balance of 2.6% was obtained, and the actual imbalance turned out to be merely 8.5%, which we interpret as an excellent result for this problem. Data replication ratio was about 6.2, less than the GPVS model's 6.7, but close enough. The near-zero load imbalance is much better than the GPVS model, which has 242% actual load imbalance for the same problem. Therefore, we think that this result confirms that our model can address the load imbalance problem adequately while not inflating the data replication, in fact it can do better in that respect as well. It turns out that the new model allows more freedom to the optimizer for both the partitioning objective and the partitioning constraint.

Chapter 5

1-D and 2-D Parallel Algorithms for All-Pairs Similarity Problem

Optimizations to the sequential all pairs similarity algorithm are covered in Section 5.1. Section 5.2 introduces the 1-D vertical and horizontal parallelizations, and likewise Section 5.3 presents the 2-D parallelization. Section 5.4 contains the performance study.

5.1 Optimizations to the sequential algorithm

In this section, we examine the optimizations in the sequential algorithms of Section 2.2.4.2 detail, as they influence our parallel algorithm design.

In our work, we have made several other versions of these algorithms to understand the impact of individual optimizations. This has aided us in understanding the advantages and disadvantages of said optimizations and design parallel algorithms. The slowness of all-pairs-2 compared to all-pairs-1 on our datasets urged us to understand the impacts of optimizations better.

all-pairs-0-array Although the input vectors are sparse, some dimensions are

dense in the real-world data that we are using. Thus, the hash table A is in fact dense. Using an array instead of a hash table improves running time.

all-pairs-0-array2 Tries to optimize all-pairs-0-array further by maintaining a list of candidate indices that are used during matching, which are zeroed before finding the matches of the next vector.

all-pairs-0-remscore remscore optimization added to all-pairs-0

all-pairs-0-minsize minsize optimization added to all-pairs-0

all-pairs-1-remscore remscore optimization added to all-pairs-1

all-pairs-1-upperbound remscore optimization added to all-pairs-1

all-pairs-1-minsize minsize optimization added to all-pairs-1

all-pairs-1-remscore-minsize minsize and remscore optimizations added to all-pairs-1

all-pairs-bruteforce Brute force algorithm that uses no intermediate data structures

The performance comparison of the various implementations on two datasets is given in Section 5.4.3, in which we see that all-pairs-0-array is the fastest implementation, therefore we focus on parallelizing that algorithm while we also discuss how to parallelize others algorithms. Note that on another software platform, perhaps one of the other variants could be as efficient as all-pairs-0-array, however, we think that the wide performance gap would be non-trivial to close.

5.2 1-D Parallel Algorithms

In the following parallel algorithms, let p be the number of processors and pid be the processor ID of the current processor. We will explain our dimension-wise and vector-wise parallelizations, respectively. We call the dimension-wise parallelization vertical, and vector-wise parallelization horizontal, for brevity and in analogy with the matrix representation D of input where each vector is a row.

5.2.1 Vertical algorithm: partitioning dimensions

In vertical parallelizations, each processor holds a number of dimensions (features), considered to be weighed by the square of number of non-zeros as for finding the matches of each vector, the entire inverted list of a dimension has to be scanned, and its contribution to the candidate matches calculated.

Each dimension d contributes

$$w[d] = |I_d|.(|I_d| + 1)/2$$

multiplications, and thus the entire work may be assumed to take $w = \sum_i w[i]$.

Since the dimensions are split across processors, each inverted list is stored wholesome. To iterate, each dimension has a home processor and each inverted list corresponding to that dimension also has the same home processor. Therefore, each processor is responsible for calculating the matches in a subspace composed of the dimensions assigned to it.

Our vertical parallel algorithms essentially parallelize the inner loop (find-matches phase) of the all-pairs-0-array algorithm, while maintaining the sequential order of processing vectors. Therefore, much attention is devoted to efficient processing of separate subspaces and merging the candidates, which is the main parallel overhead of this parallelization.

5.2.1.1 Initial distribution

The simplest distribution is cyclic distribution of dimensions, which is a random distribution of dimension, however it has turned out to result in too much load imbalance. Therefore, we use the following simple partitioning algorithm. The dimensions are sorted in order of decreasing non-zeroes and the dimensions are binned to p bins so as to balance the load. To achieve this, we use a first-fit algorithm that places the next dimension in the least loaded processor. We distribute the dimensions before starting and timing the parallel algorithm.

Algorithm 3 *Par-All-Pairs-0-Vert*$(V, t, comm)$

```
M ← ∅
I ← Make-Sparse-Matrix(m, n)
for all x = v_i ∈ V do
  M ← M ∪ Par-Find-Matches-0-Vert(x, I, t)
  for all x[j] ∈ V where x[j] > 0 do
    I_ji ← x[j]
return M
```

5.2.1.2 Inner-loop parallelization of all-pairs-0

Algorithm 3 depicts the pseudocode for the basic vertical parallelization of all-pairs-0 kind of algorithms. The *comm* variable is the MPI communicator used in the collective communication operations, it is given as a variable to make the algorithm re-usable in the 2-D algorithm. In *Par-All-Pairs-0-Vert*, first, we calculate the global number of dimensions by taking the maximum among all processors. Then, we call the parallel find-matches algorithm for each input vector x, which calculates separate candidate maps on all processors and then accumulates the candidate scores in parallel before filtering the candidates. Each processor thus computes partial candidate scores independently and synchronously. Then, scores are accumulated via collective communication, which results in each processor having a disjoint set of scores to filter, and the filtering is performed in parallel.

5.2.1.3 Local pruning optimization

We propose a local pruning optimization for the matching phase. The parallelization of the inner loop is shown in Algorithm 4. We employ local pruning to decrease the number of candidates accumulated by collective communication operations. Let us define t_{local}, the local similarity threshold.

$$t_{local} = t/p \tag{5.1}$$

Lemma 9. *Observe that, for any distribution of dimensions, if a candidate is matched, that is* $sim(x, y) \geq t$*, then the local similarity of at least one processor*

Algorithm 4 *Par-Find-Matches-0-Vert*$(x, I, t, comm)$

$t_{local} \leftarrow t/p$
$A \leftarrow$ *Make-Array*(n) such that $A[i] = 0$
for all $(x, x[i]) \in x$ **do**
  for all $(y, y[j]) \in I[i]$ **do**
    $A[y] \leftarrow A[y] + x[i].y[i]$
  $C \leftarrow \{(x, y) | A[y] \geq t_{local}\}$
  return *Accumulate-Scores-Vert*$(A, C, comm)$

should be at least t_{local}.

Proof. Assume that for all p processors, the local similarities $sim(x, y) < t_{local}$. Then, obviously, $sim(x, y) < p.t/p$, that is $sim(x, y) < t$ which is a contradiction. Therefore, on at least one processor, the local similarity is greater than t_{local}. □

Making use of this lemma, on each processor we compute the array A of local scores of x, and a set of local candidates C which are the candidates that meet local threshold t_{local} effortlessly. These local scores and candidates are then merged using a parallel score accumulation algorithm called *Accumulate-Scores-Vert*.

Note that we use arrays for candidate map instead of a hash table because it is more efficient in practice.

5.2.1.4 Score accumulation with local pruning

The scores are accumulated in two communication steps. In the first step, we perform an all-reduce operation using the binary operation of set union. At the end of this step, every processor obtains a C_g of *global* candidates. After this step, since every processor already has the local scores A, which contain all the local candidates in C_g, we take the local scores in A which are in C_g and put them into a sparse vector A'. On each processor, for each candidate vector y with weight w, we have

$$A'[y] = A[y] = w > t_{local}.$$

Succeeding that, we compute A_g which is the summation of sparse vectors on each processor, with the result partitioned over all processors, so each processor stores a range of indices of A_g. That is, we use a parallel sparse vector addition algorithm with input and output partitioning. Thereafter, the A_g can be filtered in parallel to find scores that are at least t.

5.2.1.5 Recursive local pruning

In practice, local pruning works quite effectively on two processors, but due to the nature of observed power-law like distribution of term frequencies, every binary subdivision almost doubles the number of candidates. If no local pruning is applied, we have observed that about $n/2$ candidates are required on the average. With local pruning, we observe a significant reduction of that number on two processors (about 10-fold) making the vertical partitioning competitive with horizontal partitioning.

By observing that local pruning can be applied recursively, we can decrease the communication volume of the score accumulation further. Let the dataset matrix D be vertically partitioned $\Pi(D) = \{D_1, D_2\}$ into roughly equal number of dimensions. Local pruning (Lemma 9) entails that the set of candidates is the union of all similar pairs in both sub-datasets with $t/2$. That is to say, we obtain a set of candidates by taking the union of local matches with $t/2$ threshold: $C(D,t) \supset M(D,t) = M(D_1,t/2) \cup M(D_2,t/2)$. This process can be applied recursively. For instance, another level of application would yield: $C(D_1,t/2) = M(D_{11},t/4) \cup M(D_{12},t/4)$ and $C(D_2,t/2) = M(D_{21},t/4) \cup M(D_{22},t/4)$ where $\{D_{11}, D_{12}\}$ is a vertical partition of D_1 and $\{D_{21}, D_{22}\}$ is a vertical partition of D_2.

This recursive sub-division suggests an algorithm. We first recursively partition the dimensions in k levels of recursion. At the bottom level k of recursion, we can find the matches for $M(D_p, t/2^k)$ where the dataset label p has k numerals, and communicate these pair-wise to calculate their union as the 2^{k-1} candidate sets for the higher level. Now, we must compute the matches in the higher level

to calculate the yet higher level candidates and so forth, until we have the candidates $C(D, t)$ for D. The intention here is that, instead of broadcasting all the bottom level candidates, we are communicating less. After computing candidates this way, another pass could be used for score accumulation, but interleaved execution of candidate generation and score addition steps would be faster.

In our example, consider that we have the candidates $C(D_1, t/2)$ and $C(D_2, t/2)$ after the first two candidate union operations at the bottom level 2. We need a fast method to filter these candidate sets, and the processors corresponding to D_1 must co-operate to calculate $M(D_1, t/2)$ and likewise for D_2. If the candidates are partitioned over processors in this step, the score accumulation can be performed fast in parallel. Therefore, we split the candidate set according to vector indices and communicate scores so that each processor making up D_1 has a portion of the global scores, which then it can filter to find its portion of $M(D_1, t/2)$. Note that this is also a partial score which may be useful to us later on, so we store it. Then, yet, in the top level when calculating the matches $M(D_1, t)$, score accumulation can proceed among processors with matching vector ranges of scores.

An important consideration in this algorithm is to be able to complete partial scores. For instance, a vector x may not be a candidate in D_1, D_{11} and D_{12}, but it may be a candidate in D_2. Since it wasn't a candidate in any of the candidates in the recursion sub-tree corresponding to D_1, the non-zero scores of x in D_1 would have to be added. This requires knowing which processors contributed to a score, and if there are missing we have to send requests to those processors and get the missing information. The processors in a partial sum may be represented with a bit-vector.

5.2.1.6 Functional recursive local pruning algorithm

In Algorithm 5, we give a straightforward functional algorithm for realizing recursive local pruning. We assume that we have $p = 2^k$ processors, and we apply vertical bi-partitioning recursively. The following algorithm assumes that local

scores have ben computed on each processors in array A. x is the vector for which matches are sought, and t is the similarity threshold; $i..j$ denotes an inclusive integer range.

Algorithm 5 *Merge-Scores-Rec*$(x, A, t, comm)$

```
1: pid ← MPI-Rank(comm)
2: p ← MPI-Size(comm)
3: if p=1 then
4:    M ← {(y, A[y]) | y ∈ (0...|A| − 1) ∧ A[y] > t}
5: else
6:    color ← if pid ∈ (0, p/2 − 1) then 0 else 1
7:    comm′ ← MPI-Comm-Split(comm, color, pid)
8:    M′ ← Merge-Scores-Rec(x, a, t/2, comm′)
9:    C ← Reduce-All(comm, M′, Union)
10:   AL ← {(y, A[y]) | y ∈ C ∧ A[y] > 0}
11:   AG ← Accumulate-Scores(comm, AL)
12:   M ← {(y, w) | (y, w) ∈ AG ∧ w > t}
13:   return M
```

5.2.1.7 Flat accumulation algorithm

Alternatively, we can implement *Accumulate-Scores-Vert* using the MPI Allgather function for constructing the set union of local C sets, and we can compute A_g, where each candidate vector is stored on a processor. We can compute A_g in distributed fashion by using p MPI Gather calls, and then locally adding the partial scores across dimensions. This is a practical implementation we are using in our experiments on compute clusters, however a more scalable implementation may be also developed in the future.

5.2.1.8 Hypercube accumulation algorithm

For accumulation, we can utilize an algorithm inspired by the parallel quicksort algorithm on hypercube topology. The input to the parallel accumulation algorithm is an association list of vector id, score pairs for the current vector x. Each association list is sorted in the order of vector id's. In the partition step of the

quicksort-like accumulation algorithm, the pivot is chosen as the average of random vector id's from the current subcube, and partition is made according to the vector id accordingly. After the communication step of the hypercube quicksort-like algorithm, an association list merging algorithm combines the results so that the entire association list at hand is sorted, and association pairs with identical vector id's are collapsed into a single pair with accumulated scores. In the end of the accumulation algorithm, the output association list is partitioned over the processors so that the filtering of scores is also carried out in parallel.

5.2.1.9 Processing in vector blocks

Since we process each vector separately in the basic parallelization outlined above, although the total load of each processor is balanced, a fine-grain imbalance is caused by the load imbalance of individual local score calculations of a vector x on each processor. To prevent this fine-grain imbalance problem, and also decrease the latency overhead, we process vectors not one by one, but in chunks of vectors, so that we can use a burst-mode communication. This requires also bundling the intermediate values so it naturally creates some algorithmic complexity, but in practice we have seen this to be an effective optimization for cluster architectures. Therefore we assume that this optimization has also been made.

5.2.1.10 Parallelization of partial indexing with global pruning

Par-All-Pairs-1-Vert is a parallelization of the partial indexing optimization that maintains a global bound value while deciding which dimensions to index. The partial indexing optimization is parallelized as follows. Each processor locally orders dimensions in given local V in the order of decreasing number of non-zeros along a dimension. Since each dimension has a home processor, each processor holds part of unindexed partial vectors, and calculates only a part of partial dot-products that are added to the candidate score maps. During indexing:

- An upper bound array myb is computed as prefix sum of $maxweight_i(V).x[i]$'s

in the local order of features.

- Each processor computes a global upper bound array b by summing local upper bound arrays myb's (via a multi-node accumulation to all processors)
- After matching, partially indexes the current vector x according to global upper bound during parallel matching.

During matching:

- The candidate maps on all processors are merged such that each processor has accumulated all the global candidates, written to a hash table.
- Each processor calculates local partial dot products that correspond to the vector id's in the candidate map.
- Local dot-products are merged in parallel using the hypercube score accumulation algorithm described earlier.
- Each processor adds the scores and global dot-products of the part of candidates it has been given from previous step, and filters results.

While a bit complex, this is an intuitive and correct parallelization of all-pairs-1, however suffering from some of the complexities of parallel score accumulation algorithms and the density of candidate maps.

5.2.1.11 An alternative parallelization of global pruning

An alternative parallelization of global pruning, *Par-All-Pairs-1-Vert-Alt*, violates the home processor of dimensions, and instead partitions the partial vectors blockwise so that each processor is responsible for partial vectors with consecutive vector id's. Therefore, each processor calculates a complete partial dot-product and dot-product scores do not have to be accumulated. While indexing, each unindexed partial vector is communicated to its home processor according to block partitioning of vector id's. The candidate scores are accumulated using a

variant of the hypercube score accumulation algorithm with output partitioning that knows about the complete range of vector id's so that each processor receives only the candidates it already has the partial vectors for, and the complex pivot selection is avoided.

5.2.1.12 Parallelization of partial indexing with local pruning

Par-All-Pairs-1-Vert-Local-Pruning does not maintain global upper bound values when deciding to index partially. Instead, it uses a local pruning decision, and it matches the pairs in parallel as in *Par-All-Pairs-1-Vert*. The actual matching algorithm is the same, only indexing has changed.

5.2.1.13 Parallelization of all-pairs-1-minsize

Minsize optimization does not occur much overhead when it is useless and can be implemented efficiently when distributing only dimensions. Since each inverted list has a home processor, the minsize optimization works just as well in the parallel setting, requiring no changes. For the parallel matching, we can re-use the matching step of *Par-All-Pairs-1-Vert* or another one of the two alternative implementations.

5.2.1.14 Implementation considerations

There are three main design options for *Par-All-Pairs-0-Vert*, which we shall detail now: selecting whether to implement the local pruning optimization proposed in Section 5.2.1.3, selecting the score accumulation algorithm which is either of flat accumulation or hypercube accumulation, and selecting how many vectors to process at each communication step for the block processing optimization.

We have implemented all of these different options and tested them. We implement the local pruning algorithm in the present experiments, because it is the fastest as it reduces the number of candidates considerably (an order of

magnitude), however there are some bottlenecks in the current implementation. Currently, we process in blocks of 512 vectors. Since each vector incurs memory storage for score arrays and candidate sets, we cannot process too many vectors at once. However, a large block size is beneficial for reducing the processing and communication imbalance across synchronization points.

5.2.2 Horizontal algorithm: partitioning vectors

The horizontal parallel algorithm partitions vectors instead of dimensions in the vertical algorithm. In this parallelization, we partition the vectors and then index and match in parallel without making much modification to the inner loop (matching), executing matchings in parallel over disjoint sets of vectors, however having to broadcast each vector. We distribute the vectors in cyclic fashion prior to the invocation and timing of the horizontal algorithm.

5.2.2.1 Outer loop parallelization of all-pairs-0

We now discuss how to parallelize the outer loop of all-pairs-0 kind of algorithms. In *Par-All-Pairs-0-Horiz* (Algorithm 6), each processor is given a disjoint set of vectors, i.e. each vector has a home processor. Each processor indexes only their local set of vectors; the inverted index being constructed is partitioned horizontally, aligned with the input dataset partition. We pad local list V of vectors with empty vectors so that each processor has the same number of vectors, by calculating the maximum number of vectors on a processor with a collective communication. For each iteration of the outer loop over local vectors, every processor gathers their current vector on all processors, constructing an array of vectors xa where $xa[proc]$ contains the query vector from processor $proc$. We then iterate over all processors $0..p-1$, matching the entire set of p current query vectors against the local inverted index I, using the sequential matching algorithm *Find-Matches-0* of all-pairs-0. We process in the same order on all processors to avoid redundant matches. We carefully index the local current vector only after it has been matched against the inverted index.

Algorithm 6 *Par-All-Pairs-0-Horiz*(V, t)

```
M ← ∅
I ← Make-Sparse-Matrix(m, n)
Pad V with empty vectors
for all x ∈ V do
  xa ← MPI-All-Gather(x)
  for all proc ← 0 to p − 1 do
    M ← M ∪ Find-Matches-0(xa[proc], I, t)
    if proc = pid then
      for all x[j] ∈ V where x[j] > 0 do
        I_ji ← x[j]
return M
```

5.2.2.2 Horizontal parallelization of all-pairs-1

In *Par-All-Pairs-1-Horiz*, again the matching algorithm does not need to be parallelized but calculation of global $maxweight_i(V)$ function, reordering the dimensions in order of decreasing number of non-zeroes, and calculation of other required global values like m and n have to be done in parallel. The partial indexing itself is immune from changes in the horizontal parallelization, as each processor holds as the home processor of each partial vector is the same as the vector it is derived from, and it is used in the partial dot-product on its home processor with no communication.

5.2.2.3 Horizontal parallelization of all-pairs-0-minsize and all-pairs-1-minsize

Minsize optimization can be translated to horizontal parallelization as follows. On each processor, the vectors are locally sorted in order of increasing $maxweight(x)$ for each vector x. However, the pruning has to be synchronized to the matching of all vectors to maintain correctness. This can be easily accomplished by taking the minimum of minsize's of all the current parallel matchings. We take the minimum as this conservative pruning does not introduce any errors that may be caused by choosing a higher value, ensuring that on all processors the same minsize is used, and the minimum of current minsizes in the matching algorithm

is the highest value that can be used without corrupting the result. Thus only a small modification to the sequential matching algorithm is sufficient, simply calculating a suitable global minsize value. Since the input vectors are assumed to be distributed evenly, there should not be much of a drift among their synchronous minsize values in large datasets.

5.2.2.4 Optimizations and scalability

The block processing optimization may be applied to the horizontal algorithm (*Par-All-Pairs-0-Horiz*) to improve load balance, although load balance does not suffer much in the horizontal parallelization. Initial distribution may be improved with respect to the random distribution balancing the vector sizes processed in each vector iteration of the parallel algorithm.

Compared to [74], we make use of the inverted index construction logic of all-pairs-0-array, not depending on any complex geometric data structures, and we make use of efficient collective communications of the message passing system, and provide a very sensible synchronization of processing rather than having to deal with dynamically load balancing, which results in a very elegant algorithm. Nevertheless, one of their optimizations involving bounding hyperspheres of point sets may be incorporated into the horizontal algorithm. If the vectors are initially partitioned geometrically, instead of cyclically or according to sizes, then bounding regions may be defined over each processor, and it may be possible to skip some communication and computation, although we would not expect much gain from such a computation.

The most significant parallel overhead here is the broadcast of the vectors. Therefore, there is a total communication volume of $size(V).(p-1)$ vector elements, which limits scalability in high number of processors in practice. While the intermediate data structures of the vertical algorithm have size proportional to the number of candidates in an iteration, the horizontal algorithm has to construct the entire I on each processor. We have found no simple solution to this obstacle to scalability without making a substantial re-design of the horizontal

algorithm. However, as we will see in experiments this scalability problem likely manifests itself in only large number of processors.

5.3 2-D Parallel Algorithm

Let there be a 2-D processor mesh with q rows and r columns. The two-dimensional data partitioning algorithm combines the vertical and horizontal parallelization as two respective levels of parallelization. First, we make a checkerboard partitioning of the input dataset where we distribute dimensions into r columns so as to balance load across processor columns, and we distribute vectors into q rows in cyclic order. Therefore, to each processor, we assign a set of vectors and a set of dimensions. Algorithm 7 shows the pseudocode for the *Par-All-Pairs-0-2D* algorithm. We assume in the following 2-D algorithm that *mycol* is the processor column of the current processor, *colid* is the current processor's identifier in *mycol*, *myrow* is the processor row of the current processor, and *rowid* is the current processor's identifier in *myrow*.

Algorithm 7 *Par-All-Pairs-0-2D*(V, t)

```
M ← ∅
I ← Make-Sparse-Matrix(m, n)
Pad V with empty vectors
for all x = v_i ∈ V do
  xa ← MPI-All-Gather(x, mycol)
  for all proc ← 0    to q − 1 do
    M ← M ∪ Par-Find-Matches-0-Vert(xa[proc], I, t, myrow)
    if proc = colid then
      for all x[j] ∈ V where x[j] > 0 do
        I_ji ← x[j]
return M
```

The two parallelizations can be elegantly combined by re-using the horizontal parallelization in the first level of parallelization and the vertical parallelization in the second level. Passing the *mycol* communicator to the vertical parallelization let us re-use the vertical algorithm with no modification.

Dataset	n	m	num. of non-zeroes	avg. vector size	avg. dim size	sparsity (%)
radikal	6883	136447	1072472	155.8	7.8	0.114
20-newsgroups	20001	313389	2984809	149.2	9.5	0.0476
wikipedia	70115	1350761	43285850	617.3	32.0	0.0457
facebook	66568	4618973	14277455	214.5	3.1	0.00464
virgina-tech	85653	367098	25827347	301.5	70.3	0.0821

Table 5.1: Real-world datasets used in our performance study.

The block optimization of the vertical algorithm has also been tried in our 2D experiments, but was found to cause more overhead compared to the one that does not block input vectors. In general, the implementation of the vertical algorithm was found to have a significant amount of garbage collection overhead since a lot of intermediate data is constructed and then discarded in the vertical algorithm (This accounts for about 30% of the running time). This overhead shows that there is room for improvement in the implementation of our 1-D vertical and 2-D algorithms due to the ocaml runtime overhead.

5.4 Performance Study

We first explain the datasets used and take a look at how the variants of the sequential all-pairs algorithms stack up. We have based our parallelizations on these results. Then, we demonstrate the parallel performance of our vertical, horizontal and 2D parallelizations. The parallelizations do show enough diversity in performance to justify the need for multiple parallelizations.

5.4.1 Datasets

We have based our performance study on real-world datasets, with no tuning that will make our task easier. We have made our experiments on two small and

three large such datasets, the properties of which are summarized in Table 5.1. The columns of Table 5.1 display the dataset name, number of vectors (n), number of dimensions (m), number of non-zeroes (sum of $|x|$'s), average vector size (average of $|x|$'s), average dimension size (average of $|I_d|$'s) and sparsity (number of non-zeroes divided by $n.m$), respectively.

Radikal data set contains 6893 short news articles from the website of Radikal Turkish newspaper, partitioned to 14 newspaper sections. The HTML documents were converted to text and converted to vector space representation using TFIDF weighting. 20-newsgroups is a classical text categorization dataset which consists of one thousand posts taken from 20 USENET newsgroups. The large datasets are downloaded from the Stanford WebBase Project [95]. The facebook dataset is composed of pages collected from Facebook on 09/08/2008. The wikipedia dataset is composed of pages collected from Wikipedia on 05/2006. The virginatech dataset is composed of pages collected from sites related with Virginia Tech shooting on 04/23/2007.

5.4.2 Implementation details

We have implemented the algorithms using Ocaml programming language 3.12, and the Ocaml MPI bindings for communication.

Since Ocaml does not have 32-bit floating point values, we resorted to a fixed point implementation that uses 32 bits to store numbers and reserves a number of fixed bits to integer and decimal point parts. In very few cases there is some loss of accuracy which causes some pairs to be missed but that is insignificant enough that we will not analyze it.

We have in general paid attention to low-level issues and used fast data structures such as arrays and lists where applicable. For the hash tables in candidate maps of original all-pairs-0, all-pairs-1, and all-pairs-2 algorithms, we used Ocaml's Hashtbl implementation in the standard library. We initialize the hash table with one fourth of the number of vectors. For document vectors we used

Algorithm	$t = 0.2$	$t = 0.3$	$t = 0.4$
all-pairs-0	141.62	142.34	143.14
all-pairs-0-array	*24.57*	*24.44*	*24.74*
all-pairs-0-array2	29.50	29.37	29.53
all-pairs-0-remscore	180.21	179.75	180.41
all-pairs-0-minsize	149.37	149.70	149.43
all-pairs-1	87.10	79.05	71.73
all-pairs-1-array	73.02	69.54	64.40
all-pairs-1-remscore	180.55	181.79	182.02
all-pairs-1-upperbound	200.96	171.42	145.90
all-pairs-1-minsize	89.57	80.52	72.21
all-pairs-1-remscore-minsize	93.31	82.70	73.42
all-pairs-2	198.92	165.64	138.98
all-pairs-bruteforce	183.06	183.32	183.28

Table 5.2: Sequential running time on radikal dataset

compressed row storage, on arrays. For inverted lists we used dynamically sized vectors with a fast way ($O(1)$) to pop the beginning of the inverted list, which is required by the minsize optimization.

5.4.3 Sequential performance

Table 5.2 shows the running times of various sequential all-pairs algorithms on radikal dataset with a few meaningful support thresholds. Likewise, Table 5.3 shows the running time on the 20-newsgroups dataset. The dot-product thresholds were chosen so that we obtain roughly $n.\lg(n)$ pairs for n vectors and increase the threshold until we have about n pairs. $n.\lg(n)$ pairs should be sufficient to construct a well connected epsilon neighborhood graph, given each vector has about $\lg(n)$ neighbors, since it is well-known that to establish inter-cluster connectivity setting $k \sim \log(n)$ is the lowest sufficient rate for knn graphs [96].

We have had to compare the effects of different optimization strategies so that we could determine which algorithm could be parallelized best. In case, there is a clearly best algorithm, the other parallelizations would be redundant. Not all optimization strategies may be parallelized well, either. The effect of optimizations may also depend on the software and hardware platform used. Therefore,

Algorithm	$t = 0.4$	$t = 0.5$	$t = 0.6$
all-pairs-0	2887.1	2904.7	2900.9
all-pairs-0-array	*480.0*	*495.4*	*478.0*
all-pairs-0-array2	596.4	618.0	595.6
all-pairs-0-remscore	3482.2	3486.1	3501.2
all-pairs-0-minsize	3171.6	3180.7	3191.4
all-pairs-1	1169.0	1035.6	882.9
all-pairs-1-array	883.8	837.0	757.5
all-pairs-1-remscore	3502.0	3499.9	3497.7
all-pairs-1-upperbound	1940.8	1649.5	1410.1
all-pairs-1-minsize	1190.0	1023.5	902.6
all-pairs-1-remscore-minsize	1288.5	1086.4	943.6
all-pairs-2	1733.8	1450	1190.2
all-pairs-bruteforce	1866.3	1867.2	1871.5

Table 5.3: Sequential running time on 20-newsgroups dataset

we made variations on the original all-pairs-0 algorithm so that optimizations were tested individually and in combination. Surprisingly, it turns out that the best algorithm is an optimization of all-pairs-0 itself that uses arrays instead of hash tables for score accumulation, named all-pairs-0-array in the tables. The running times show that it is actually quite difficult to improve upon the brute force algorithm. While that may sound frustrating, it also means that there is positive research potential in designing new all pairs similarity algorithms, since it is known that the optimal nearest neighbor query algorithms in R^d have low running time complexity [80]. With some work, those results could carry over to real-world data with high dimensionality and sparse vectors.

Algorithm all-pairs-0-array2 fares worse than all-pairs-0-array unexpectedly, the only difference in the former is maintaining a list so that the written entries can be zeroed out in the next iteration. It seems that this list maintenance and zeroing only written entries is more expensive than zeroing out the entire array, suggesting that the non-zero entries are too many for this optimization. We also see that, somewhat unexpectedly, all-pairs-2 does not improve on all-pairs-1. However, the minsize optimization over all-pairs-1 improves on all-pairs-1, which is part of all-pairs-2. The other two optimizations of all-pairs-2 (all-pairs-1-remscore and all-pairs-1-upperbound) apparently slow down all-pairs-1 instead

of accelerating it. All-pairs-1-remscore-minsize is also worse than all-pairs-1-minscore, suggesting that the remscore optimization is not useful at all. All-pairs-2 is almost as slow as bruteforce for lower thresholds, so it may not be a very meaningful algorithm to study. Even at higher thresholds, where there are too few outputs, the results do not change significantly. Still, the most interesting result is that, all-pairs-0-array is much better than all of those optimizations. The array optimization carries over to all-pairs-1, but all-pairs-1-array still does not match all-pairs-0-array, so we did not see a need to try to apply it to other variants. Likewise, the optimizations over all-pairs-0 (all-pairs-0-remscore, and all-pairs-0-minsize) seem to be worse than the brute force algorithm therefore they were not worth pursuing.

While in this work, we focus on the parallelization of existing algorithms and their variants, we have also examined the real reason for the high running times. The first reason is that there are a lot of dimensions, breaking down easy separability of points, and second and most importantly that the density of the dimensions follow a power-law distribution which introduces an almost irreducible complexity in the processing of the densest dimensions. The reason why partial indexing optimization is more effective than the other optimizations is that it separates the processing into a dense and a sparse phase, where a brute force algorithm is applied to the dense part of the data and an indexing approach is applied to the sparse part, definitely improving over the plain brute force algorithm especially in the case of higher thresholds as can be seen in the running times. Still, the improvement seems to be on a constant order, which is interesting as it suggests that there is no asymptotic improvement.

5.4.4 Parallel performance

Our experiments were carried out at the TUBITAK ULAKBIM High Performance Computing Center, which is comprised of 48-core multi-processor nodes built with AMD Opteron 6172 processors, interconnected with an Infiniband network, running GNU/Linux operating system. In the rest of the thesis, we use processor and core interchangeably. For each dataset, we worked on a single, meaningful

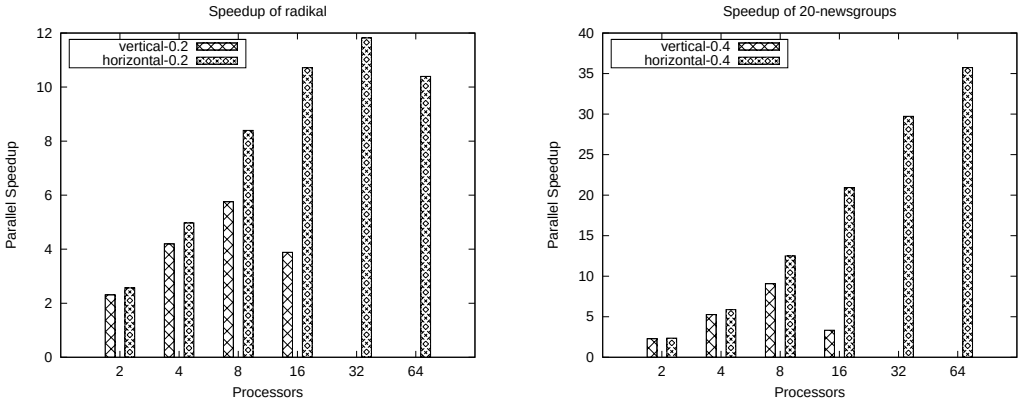
Dataset	t	Time	Matches
radikal	0.2	15.5	16810
20-newsgroups	0.4	317.3	64396
wikipedia	0.9	54424.0	747999
facebook	0.99	10777.8	819196
virgina-tech	0.99	10426.2	13447874

Table 5.4: The problem instances used in our study

similarity threshold. The similarity thresholds for each dataset were chosen so that they would result in a well-connected similarity graph. We again followed the notion of allowing about $n \log n$ similar pairs in the output as a rough guideline, we made sure that we obtained a significant number of similar pairs. Table 5.4 shows the problem instances used in our parallel performance study; the columns display the similarity threshold t, the running time of the sequential algorithm all-pairs-0-array and the number of similar pairs output. For 1-D algorithms, we ran our algorithms up to 256 processors on the small datasets and up to 64 processors on the large datasets due to resource limitations on the batch system of the supercomputer (128 in one instance where we could run it). For the 2-D algorithms, we have tried different combinations of numbers of processor rows and columns in the virtual mesh, again going up to 256 at most for small datasets and 128 processors at most for large datasets.

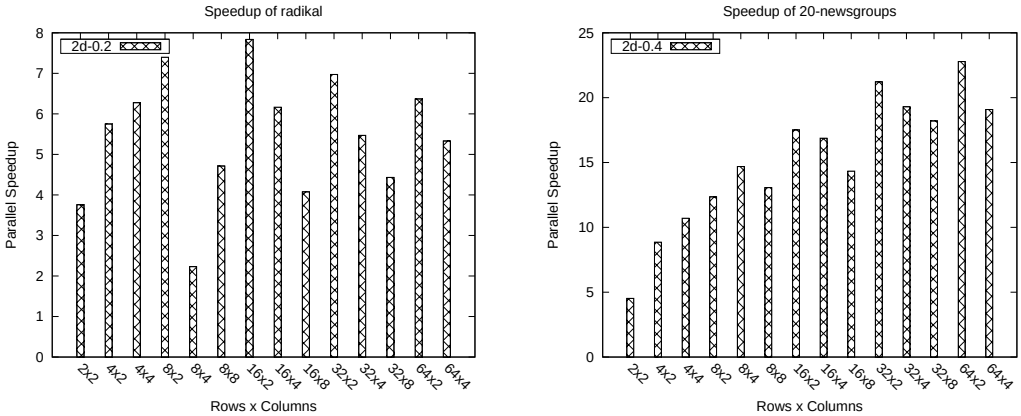
We have performed our experiments on three algorithms. The vertical parallel algorithm is Algorithm 3 that uses the major optimization of score accumulation with local pruning (Lemma 9). Otherwise, it is not possible to get good speedup beyond 2 cores, the results of which would clutter the results and would be infeasible for most of the experiments. We use the flat score accumulation algorithm in the experiments (Section 5.2.1.7), the other choices were covered in Section 5.2.1.14. The horizontal parallel algorithm is Algorithm 6, explained in Section 5.2.2.

Fig. 5.1 shows the parallel speedup of our vertical and horizontal algorithms on the smaller two datasets: radikal and 20-newsgroups. Likewise, Fig. 5.2 shows the speedup of our 2-D algorithm on the same datasets. Similarly, Fig. 5.3 depicts the parallel speedup of our 1-D algorithms on the larger three datasets: wikipedia,



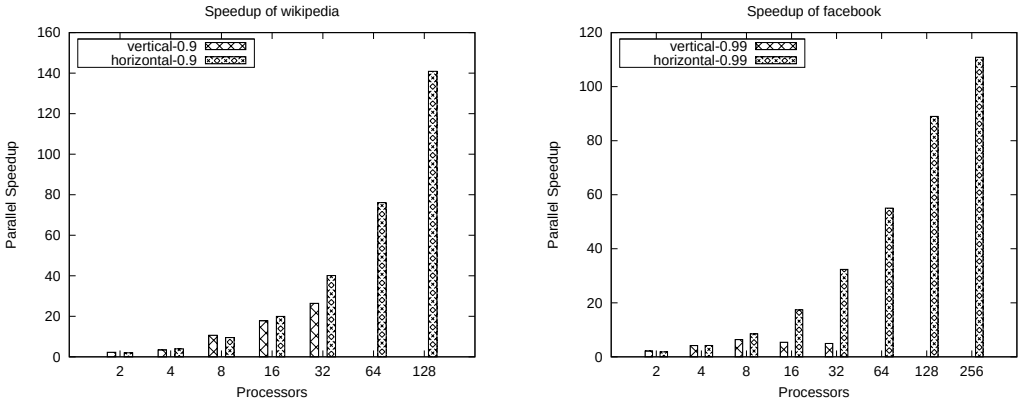
(a) radikal (b) 20-newsgroups

Figure 5.1: Parallel speedup of horizontal and vertical algorithms on small datasets radikal and 20-newsgroups



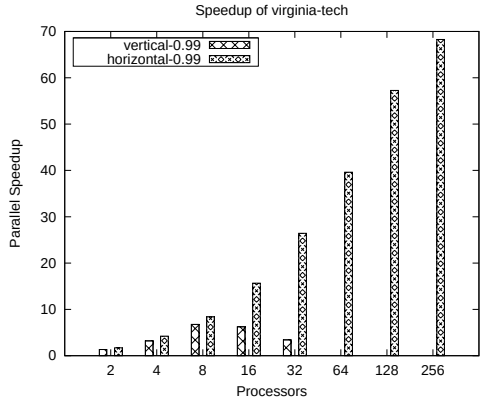
(a) radikal (b) 20-newsgroups

Figure 5.2: Parallel speedup of the 2D algorithm on small datasets radikal and 20-newsgroups



(a) wikipedia

(b) facebook

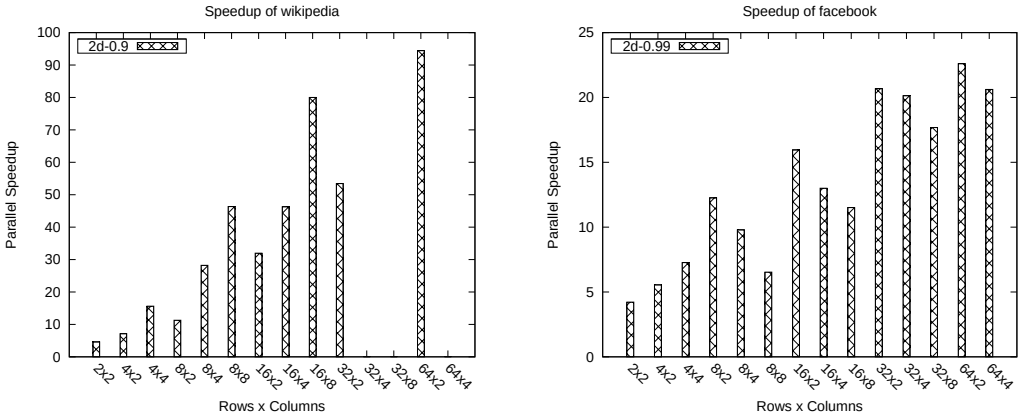


(c) virginia-tech

Figure 5.3: Parallel speedup of horizontal and vertical algorithms on the large datasets: wikipedia, facebook, virginia-tech

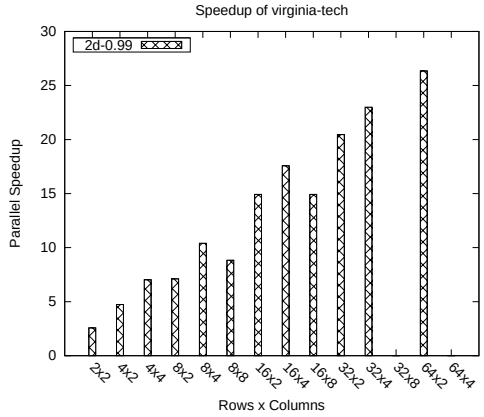
facebook and virgina-tech, while Fig. 5.4 gives the speedups of the 2-D algorithm for the same three datasets. The processor configurations of the 2D algorithm are indicated as $p \times q$ on the x-axis where p is the number of the processor rows and q is the number of processor columns.

In all datasets, we see that the horizontal algorithm scales better than the vertical algorithm. The vertical algorithm scales well up to 8 processors, but after that it loses quite a bit of steam. It is still quite an achievement that the vertical algorithm scales as much, since the number of processors increase the



(a) wikipedia

(b) facebook



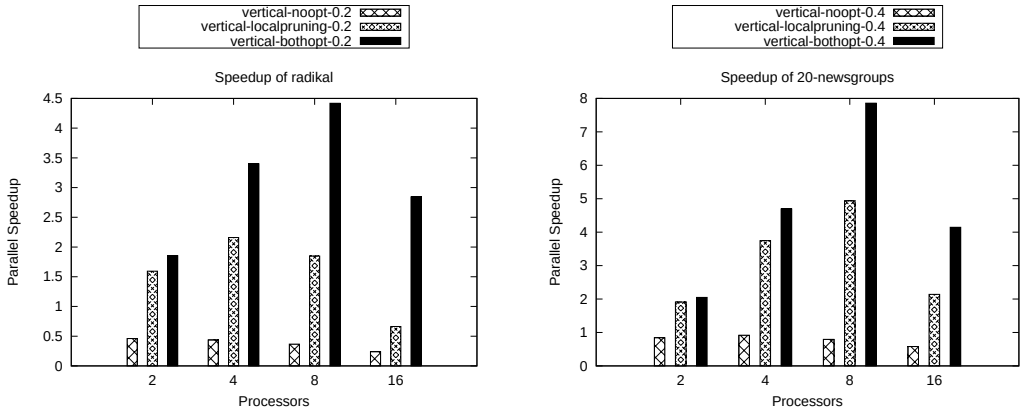
(c) virginia-tech

Figure 5.4: Parallel speedup of the 2D algorithm on the large datasets: wikipedia, facebook, virginia-tech

communication volume and communication asynchrony rapidly despite the local pruning optimization. The horizontal algorithm scales well up to 32 processors and then starts to slow down due to the fact that the broadcast starts becoming significant. This is most apparent in radikal dataset, but it is also seen in other datasets that the speedup does not accelerate as much, as we go up to 64 processors. We observe that both vertical and horizontal parallelizations achieve super-linear speedups in several cases, affirming the efficiency of our implementation, as in those cases the algorithms make better use of the memory hierarchy. In two cases, we see that the vertical algorithm achieves better speedup than the

horizontal algorithm, justifying the usefulness of our vertical algorithm. The 2-D algorithm shows varying performance according to the processor configuration. Since the vertical algorithm did not scale further than 8 processors, we did not try more processor columns in the virtual mesh. We sometimes see excellent speedups with the 2-D algorithm, for instance in wikipedia, 4×4 yields super-linear speedup and 16×8 yields about 80 speedup. However, on the average, the 2-D algorithm's performance is between that of the horizontal and vertical algorithms, it is usually better than half of the speedup of the horizontal algorithm for the maximum number of processors although for facebook dataset it's slightly worse than that.

5.4.5 Local pruning and block processing optimizations

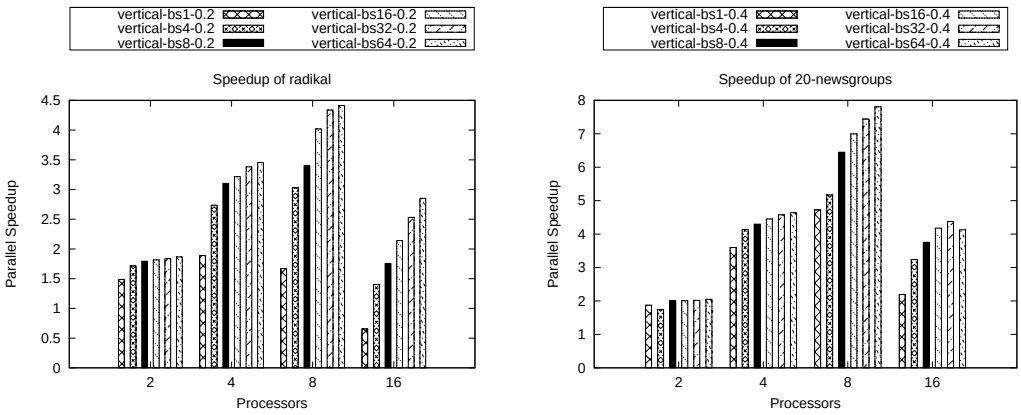


(a) radikal

(b) 20-newsgroups

Figure 5.5: Speedup comparison of three parallel algorithms on radikal and 20-newsgroups datasets

It is useful to understand the performance impact of local pruning and block processing optimizations for the vertical algorithm. Without those optimizations, the vertical algorithm is futile, it would not be quite possible to apply it to sufficiently many cases. Therefore, we show its performance, when neither optimization is applied, and when only local pruning is applied, together with happens when both optimizations are turned on.



(a) radikal

(b) 20-newsgroups

Figure 5.6: Speedup comparison of varying block sizes on radikal and 20-newsgroups datasets

We have chosen the smaller two datasets radikal and 20-newsgroups for this comparison, because some of the runs would be infeasible for the larger datasets. We run them on up to 16 processors, which is sufficient to illustrate the performance differences. Fig. 5.5 shows how speedup varies for different vertical algorithms on small datasets, comparing the unoptimized vertical algorithm (vertical-noopt), the vertical algorithm with local pruning optimization only (vertical-localpruning) and the vertical algorithm with both local pruning and block processing optimizations applied (vertical-bothopt). It is clearly seen that local pruning improves over no optimization and both optimizations together improve on local pruning only. Local pruning is more significant for smaller number of processors and block processing is more significant for larger number of processors. In fact, without these optimizations, we see that it would be impossible to get a speedup greater than 1 on any number of processors. It is only due to these effective optimizations that we have been able to obtain the speedups previously demonstrated. The comparison is similar across two datasets. The optimizations are most effective on 8 processors; on 16 processors, the effectiveness of the local pruning algorithm declines greatly, which is why we did not extend the study to a larger number of processors.

Comparing speedups alone does not give us much insight into how these speed differences occur. We have thus profiled the algorithms in detail. We have measured the time elapsed for both communication and computation phases in the algorithms. We have also calculated how the number of candidates vary when we use the optimizations, and how many scores are actually accumulated. We have also put a barrier before each collective communication operation, so that we can measure how much processors wait before engaging in actual communication. We give both average and maximum values for the measured values, to show how imbalance for these values vary. In Table 5.5, Table 5.6, Table 5.7, and Table 5.8 we measure the following parameters for varying number of processors and algorithms: p shows the number of processors, Algo. shows the algorithm being used, C_{avg} shows the average time of communication, C_{max} shows the maximum time of communication, W_{avg} shows the average time of work, W_{max} shows the maximum time of work, Scores shows the total number of scores communicated, $Cand_{avg}$ shows the average number of candidates, $Cand_{max}$ shows the maximum number of candidates $Barr_{avg}$ shows the average barrier time, $Barr_{max}$ shows the maximum barrier time.

Table 5.5 and Table 5.6 show the profiling results for the three vertical algorithm variants previously mentioned. The profiling data suggests that the local pruning optimization is effective for reducing communication time, and the number of scores communicated. On 2 processors, we see that it reduces more than 100-fold. Even on 16 processors, there is a 10-fold improvement on the number of scores communicated. The work time is also reduced due to fewer scores being processed. The barrier time also reduces favorably for local pruning optimization. However, block processing further reduces barrier time, and consequently, the communication time. It turns out that block processing optimization is very effective for the all-pairs similarity problem, as otherwise the effects of small communication latencies and imbalances must be aggregating. We see that the work time slightly increases, but this is offset by the huge savings in communication time. For instance, on 8 processors, for 20-newsgroups dataset, the maximum communication time reduces from 31.22 seconds to 7.02 seconds, while maximum work time increases from 24.34 seconds to 25.89 seconds, and the barrier time

reduces from 15.87 to 5.73. These are quite significant savings for a parallel algorithm.

Table 5.7 and Table 5.8 show the profiling results when only the processing block size is varied in the fully optimized vertical algorithm, where algorithm "vertical-bsx" means a block size of x. We see that, generally, enlarging block size improves reduction of communication time and barrier time. The communication imbalances also follow a decreasing trend as the block size increases, which shows that our statistical reasoning works. Especially, the communication times become much more even as the block size is increased. The barrier time also follows a similar trend, but it does not become as finely balanced. The communication and barrier times are very small already even for a block size of 64, so more intelligent document partitioning methods may not be very effective in improving communication performance. We did not increase the block size much further, since every document in the block incurs a large memory penalty. We did get out of memory errors with a block size of 256 on larger datasets. In general, the block size must be specified with the dataset size in mind so as to prevent such errors.

p	Algo.	C_{avg}	C_{max}	W_{avg}	W_{max}	$Barr_{avg}$	$Barr_{max}$	Scores	$Cand_{avg}$	$Cand_{max}$
2	vertical-noopt	12.03	12.09	8.63	8.71	3.15	4.42	23684403	0.0	0
2	vertical-localpruning	1.27	1.32	5.81	5.86	0.74	0.81	42086	22886.5	23272
2	vertical-bothopt	0.04	0.04	6.26	6.36	0.24	0.34	42086	22886.5	23272
4	vertical-noopt	18.41	18.72	4.12	4.28	6.69	9.98	23684403	0.0	0
4	vertical-localpruning	2.34	2.38	2.78	2.87	0.97	1.04	116000	34393.8	38986
4	vertical-bothopt	0.10	0.12	3.17	3.25	0.25	0.27	116000	34393.8	38986
8	vertical-noopt	27.15	27.91	2.02	2.16	11.47	17.21	23684403	0.0	0
8	vertical-localpruning	4.55	4.60	1.55	1.65	1.51	1.93	355937	53711.8	73642
8	vertical-bothopt	0.41	0.51	1.87	2.02	0.42	0.60	355937	53711.8	73642
16	vertical-noopt	47.35	48.04	1.21	1.55	10.57	13.52	23684403	0.0	0
16	vertical-localpruning	17.07	17.36	0.93	1.06	1.69	2.90	1155714	89717.0	202112
16	vertical-bothopt	2.42	2.57	1.23	1.35	0.54	0.89	1155714	89717.0	202112

Table 5.5: Profiling of vertical variants on radikal dataset

p	Algo.	C_{avg}	C_{max}	W_{avg}	W_{max}	$Barr_{avg}$	$Barr_{max}$	Scores	$Cand_{avg}$	$Cand_{max}$
2	vertical-noopt	123.08	124.63	153.79	155.23	34.17	44.30	194138198	0.0	0
2	vertical-localpruning	12.84	14.91	134.68	136.86	10.84	12.84	287786	148376.0	246016
2	vertical-bothopt	0.17	0.18	137.88	139.72	3.29	5.13	287786	148376.0	246016
4	vertical-noopt	177.56	178.94	78.42	79.00	70.60	99.58	188179681	0.0	0
4	vertical-localpruning	18.67	19.49	54.76	56.04	14.04	14.55	1060564	274885.0	398405
4	vertical-bothopt	0.81	1.04	56.62	57.79	4.03	4.79	1060564	274885.0	398405
8	vertical-noopt	266.82	274.48	42.87	62.64	114.28	158.89	180315935	0.0	0
8	vertical-localpruning	30.78	31.22	23.33	24.34	13.79	15.87	4165217	551323.0	1939290
8	vertical-bothopt	6.19	7.02	25.22	25.89	4.83	5.73	4165217	551323.0	1939290
16	vertical-noopt	434.75	440.22	22.93	23.98	120.81	144.17	172874767	0.0	0
16	vertical-localpruning	111.18	111.83	16.82	18.08	13.58	17.53	17454734	1203360.0	11294606
16	vertical-bothopt	47.05	48.16	15.53	19.16	5.40	6.87	17454734	1203360.0	11294606

Table 5.6: Profiling of vertical variants on 20-newsgroups dataset

p	Algo.	C_{avg}	C_{max}	W_{avg}	W_{max}	$Barr_{avg}$	$Barr_{max}$	Scores	$Cand_{avg}$	$Cand_{max}$
2	vertical-bs1	0.73	0.74	6.48	6.61	0.90	0.99	42086	22886.5	23272
2	vertical-bs4	0.23	0.23	6.36	6.47	0.52	0.61	–	–	–
2	vertical-bs8	0.12	0.13	6.35	6.38	0.34	0.36	–	–	–
2	vertical-bs16	0.08	0.08	6.33	6.45	0.34	0.45	–	–	–
2	vertical-bs32	0.05	0.05	6.32	6.34	0.29	0.31	–	–	–
2	vertical-bs64	0.04	0.04	6.29	6.33	0.23	0.25	–	–	–
4	vertical-bs1	1.78	1.87	3.40	3.48	1.34	1.43	116000	34393.8	38986
4	vertical-bs4	0.50	0.52	3.29	3.36	0.67	0.76	–	–	–
4	vertical-bs8	0.28	0.30	3.22	3.29	0.42	0.47	–	–	–
4	vertical-bs16	0.20	0.23	3.20	3.29	0.36	0.41	–	–	–
4	vertical-bs32	0.14	0.16	3.17	3.26	0.27	0.31	–	–	–
4	vertical-bs64	0.10	0.11	3.16	3.26	0.24	0.28	–	–	–
8	vertical-bs1	3.61	3.91	2.02	2.16	1.94	2.27	355937	53711.8	73642
8	vertical-bs4	1.10	1.24	1.94	2.06	1.03	1.21	–	–	–
8	vertical-bs8	0.73	0.87	1.92	2.07	0.94	1.13	–	–	–
8	vertical-bs16	0.50	0.64	1.90	2.04	0.60	0.81	–	–	–
8	vertical-bs32	0.40	0.51	1.90	2.03	0.47	0.65	–	–	–
8	vertical-bs64	0.41	0.52	1.88	2.03	0.42	0.60	–	–	–
16	vertical-bs1	15.39	16.12	1.37	1.53	2.73	3.71	1155714	89717.0	202112
16	vertical-bs4	6.39	6.66	1.31	1.45	1.17	1.69	–	–	–
16	vertical-bs8	4.77	4.95	1.29	1.42	0.97	1.40	–	–	–
16	vertical-bs16	3.57	3.74	1.28	1.43	0.85	1.25	–	–	–
16	vertical-bs32	2.85	3.01	1.28	1.44	0.64	1.00	–	–	–
16	vertical-bs64	2.43	2.58	1.23	1.34	0.55	0.89	–	–	–

Table 5.7: Profiling of various block sizes on radikal dataset

p	Algo.	C_{avg}	C_{max}	W_{avg}	W_{max}	$Barr_{avg}$	$Barr_{max}$	Scores	$Cand_{avg}$	$Cand_{max}$
2	vertical-bs1	2.66	2.69	138.96	140.97	11.56	13.88	287786	148376.0	246016
2	vertical-bs4	0.88	0.88	148.88	158.13	15.14	24.28	–	–	–
2	vertical-bs8	0.50	0.52	137.95	140.12	5.35	7.53	–	–	–
2	vertical-bs16	0.30	0.31	138.80	140.88	4.54	6.64	–	–	–
2	vertical-bs32	0.39	0.40	138.46	140.61	3.96	6.10	–	–	–
2	vertical-bs64	0.17	0.18	137.19	138.65	3.09	4.54	–	–	–
4	vertical-bs1	5.87	6.25	58.95	59.57	15.29	16.31	1060564	274885.0	398405
4	vertical-bs4	2.00	2.14	58.06	58.83	9.44	10.42	–	–	–
4	vertical-bs8	1.51	1.76	57.70	59.07	7.69	8.60	–	–	–
4	vertical-bs16	0.92	1.01	57.32	57.89	6.27	7.22	–	–	–
4	vertical-bs32	0.78	0.89	57.47	58.92	4.55	5.55	–	–	–
4	vertical-bs64	0.81	1.04	57.00	58.15	4.12	4.94	–	–	–
8	vertical-bs1	17.20	18.70	27.30	27.84	16.77	18.05	4165217	551323.0	1939290
8	vertical-bs4	11.28	12.47	28.66	35.00	15.51	18.36	–	–	–
8	vertical-bs8	8.61	9.55	26.53	27.17	9.16	9.90	–	–	–
8	vertical-bs16	6.93	7.94	26.06	26.72	7.72	8.90	–	–	–
8	vertical-bs32	6.62	7.55	25.71	26.30	5.87	6.83	–	–	–
8	vertical-bs64	6.19	7.02	25.30	25.97	4.84	5.70	–	–	–
16	vertical-bs1	92.72	96.26	20.55	21.68	18.74	21.27	17454734	1203360.0	11294606
16	vertical-bs4	59.00	61.03	18.62	20.42	10.41	12.41	–	–	–
16	vertical-bs8	50.25	52.13	17.44	19.17	7.88	9.74	–	–	–
16	vertical-bs16	44.52	46.23	16.49	18.98	6.60	8.32	–	–	–
16	vertical-bs32	43.04	44.83	15.89	18.67	5.35	7.05	–	–	–
16	vertical-bs64	47.19	48.26	15.52	19.02	5.30	7.00	–	–	–

Table 5.8: Profiling of various block sizes on 20-newsgroups dataset

Chapter 6

Conclusion

We have introduced three new coarse-grain data-parallel frequent itemset mining (FIM) algorithms using a top-down data partitioning scheme with selective replication. Section 6.1 summarizes the results with the first two of these algorithms that use a graph partitioning by vertex separator (GPVS) model, and Section 6.2 mentions the results obtained from the third algorithm that uses the hypergraph partitioning model. Section 6.3 briefly discusses the conclusions from our $1-D$ and $2-D$ algorithms for the parallel all pairs similarity problem, and Section 6.4 exposes future work related to the thesis.

6.1 NoClique and NoClique2 methods

We have proposed a novel divide-and-conquer strategy suitable for parallelization of the FIM task. Our objective is to divide the whole transaction database into parts that can be mined independently. It turns out that we can distribute items so as to achieve our goal of independent mining, while replicating some items selectively, implying an amount of work that cannot be divided further in the same fashion. This optimization problem is cast as a Graph Partitioning by Vertex Separator (GPVS) problem where the partitioning objective corresponds to minimizing data replication or collective work (work that requires collective

communication) by setting appropriate weights to vertices, and the partitioning constraint corresponds to maintaining storage balance or computational load by setting appropriate weights likewise. The transaction database distribution is independent of the underlying database representation and the serial mining algorithms employed. We have proposed an item distribution method that depends on theoretical observations that identify lack of cliques among two sets of items in G_{F_2}. The mining problem is decomposed into independent sub-problems using a graph partitioning by vertex separator (GPVS) model which encapsulates the minimization of task or data redundancy as well as computational load or storage balance. We showed that this model can be extended to n-way distribution and any level of mining. Based on our distribution model, we designed and implemented two parallel FIM algorithms called *NoClique* and *NoClique2*. *NoClique* algorithm is a black-box parallelization with redundant work that can parallelize any given serial FIM algorithm and *NoClique2* is a parallel vertical mining algorithm based on our new serial mining algorithm called *Bitdrill*. Experiments with synthetic and real-world databases on a Beowulf cluster showed that our method has competitive performance with respect to a state-of-the-art parallel frequent itemset mining implementation.

6.2 Intelligent Candidate and Item Distribution method

We have addressed the load imbalance problem that we have observed in *NoClique2* and *NoClique2* by means of a new model. We also introduce a hypergraph partitioning model for the parallel FIM task which improves the load imbalance of the GPVS model. This model gives rise to a simple but powerful parallel algorithm called Intelligent Candidate and Item Distribution *ICID* that distributes both candidates and items, optimizing the amount of communication and balancing the load in a fine-grain manner. We also introduce a re-partitioning model because only a few levels can be mined in parallel with the new algorithm, which requires the candidates for the next few levels to be generated in advance.

Experiments have shown that the new parallel FIM algorithm does have much better load imbalance, and that it surpasses the performance of *NoClique2* in some cases.

6.3 Parallel All Pairs Similarity

We have designed new parallel algorithms for the efficient practical algorithms proposed by Bayardo et. al [14]. We have compared various optimizations to the practical algorithms, and we have found that a simple optimization to all-pairs-0 which we call all-pairs-0-array gave the best results. We have been able to distribute both vectors and dimensions in a way that is faithful to the original processing order and data structures of all-pairs-0-array. The vertical parallel algorithm distributes dimensions and parallelizes the inner loop, accumulating candidates. We have proposed an effective pruning step to decrease the number of candidates communicated in this step (Lemma 9). Various optimizations and implementation choices for the vertical algorithm have been considered, including a recursive similarity match search algorithm. The horizontal parallel algorithm is easier and it parallelizes the outer loop of the algorithm. We have also proposed a 2-D parallel algorithm which combines the inner-loop and outer-loop parallelizations in an elegant fashion. Our experiments show that the variety of parallelizations is useful for large-scale similarity graph construction.

6.4 Future Work

In the future, more appropriate optimization models may be developed for the parallel FIM problem following our approach. For instance, the partitioning constraint of GPVS may be modified such that the balance is defined on the weights of the parts together with the separator rather than just those of parts. Note that to optimize data and task redundancy at the same time, we may need a multi-objective GPVS tool which uses multiple weights per vertex. Using a

parallel GPVS tool or a parallel hypergraph partitioning tool that is suitable for hypergraph-based GPVS solution would be necessary for an efficient implementation. With regards to performance of the GPVS algorithm, FIM specific optimizations may enhance the performance of GPVS. Since the effectiveness of our approach depends on sufficient sparsity of G_{F_l} for a given level l, it may also be possible to automate the selection of parameters, for instance l or the support threshold. We have observed that the performance of the mining algorithm depends on the probability distribution of items in the database, which suggests that statistically aware parallel mining programs may be useful. We also think that efficient and scalable parallel FIM algorithms (following the design in [9] for instance) that work on terabyte scale databases are needed along with extensive benchmarks on real-world databases.

Our *ICID* algorithm may be interpreted as an enhancement of *Candidate-Distribution*. It seems logical to try to improve other previous parallel algorithms. *ICID* may be implemented better with full parallelization of the candidate generation and PaToH steps, and experiments may include full mining results with the re-partitioning model. Several optimizations are conceivable for *ICID*, for instance the mining problem may be modified so that there will be fewer candidates and frequent itemsets, or candidates may be clustered appropriately so that blocks of candidates are represented by a single vertex, speeding up hypergraph partitioning step.

In the future, we would like to see if we can incorporate more techniques to prune candidates, or other optimizations into our framework for parallel all-pairs similarity algorithms. For instance, it may be possible to exploit the Zipf-like distribution of dimension frequencies, in a better way. Or certain data decomposition approaches, like that of [68], may be incorporated. It may also be worthwhile to investigate the applicability of our data distribution approach to approximate similarity search and knn algorithms, as well as different algorithmic approaches to proximity search. The scalability of both the vertical and the 2-D algorithms could be improved upon. For the vertical algorithm, a better recursive local pruning algorithm could be useful, or more intelligent pruning heuristics could be discovered. For the 2-D algorithm, a better implementation could make use

of asynchronous communication and burst-mode transfers. In general, it is an open problem to find the best data decomposition for parallel solutions of this problem which does not suffer from the replication bottleneck of the horizontal distribution. Our present results may lead to better solutions in that area, eventually.

Bibliography

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 487–499, Morgan Kaufmann Publishers Inc., 1994.

[2] M. J. Zaki, "Generating non-redundant association rules," in *Knowledge Discovery and Data Mining*, pp. 34–43, 2000.

[3] D.-I. Lin and Z. M. Kedem, "Pincer search: A new algorithm for discovering the maximum frequent set," in *6th Intl. Conf. Extending Database Technology*, pp. 105–119, 1998.

[4] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Trans. On Knowledge And Data Engineering*, vol. 8, pp. 962–969, 1996.

[5] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "Parallel algorithms for discovery of association rules," *Data Mining and Knowledge Discovery*, vol. 1, no. 4, pp. 343–373, 1997.

[6] D. W. Cheung, V. T. Ng, A. W. Fu, and Y. J. Fu, "Efficient mining of association rules in distributed databases," *IEEE Trans. On Knowledge And Data Engineering*, vol. 8, pp. 911–922, 1996.

[7] O. Zaïane, M. El-Hajj, and P. Lu, "Fast parallel association rule mining without candidacy generation," in *Proc. of the IEEE 2001 International Conference on Data Mining (ICDM'2001)*, (San Jose, CA, USA), November 29–December 2 2001.

[8] A. Rudra, R. P. Gopalan, and Y. G. Sucahyo, "Scalable parallel mining for frequent patterns from dense datasets using a cluster of pcs," in *Proceedings of Sixth International Conference on Information Technology*, (Bhubaneswar, India), Dec 22–25 2003.

[9] E.-H. Han, G. Karypis, and V. Kumar, "Scalable parallel data mining for association rules," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, pp. 337–352, May 2000.

[10] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, *High Performance Computing for Computational Science - VECPAR 2002: 5th International Conference, Porto, Portugal, June 26-28, 2002. Selected Papers and Invited Talks*, vol. 2565, ch. An Efficient Parallel and Distributed Algorithm for Counting Frequent Sets. Springer, 2003.

[11] C. Lucchese, S. Orlando, and R. Perego, "Fast and memory efficient mining of frequent closed itemsets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 1, 2006.

[12] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *RecSys* (P. Pu, D. G. Bridge, B. Mobasher, and F. Ricci, eds.), pp. 107–114, ACM, 2008.

[13] A. Savasere, E. Omiecinski, and S. B. Navathe, "An efficient algorithm for mining association rules in large databases," in *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pp. 432–444, 1995.

[14] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of the 16th international conference on World Wide Web*, WWW '07, (New York, NY, USA), pp. 131–140, ACM, 2007.

[15] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *2000 ACM SIGMOD Intl. Conference on Management of Data* (W. Chen, J. Naughton, and P. A. Bernstein, eds.), pp. 1–12, ACM Press, May 2000.

[16] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (P. Buneman and S. Jajodia, eds.), (Washington, D.C.), pp. 207–216, 26–28 1993.

[17] R. Agrawal and R. Srikant, "Mining sequential patterns," in *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 1995.

[18] S. Brin, R. Motwani, and C. Silverstein, "Beyond market baskets: Generalizing association rules to correlations," in *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pp. 265–276, ACM Press, 1997.

[19] C. Silverstein, S. Brin, and R. Motwani, "Beyond market baskets: Generalizing association rules to dependence rules," *Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 39–68, 1998.

[20] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, pp. 259–289, 1997.

[21] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13–15, 1997, Tucson, Arizona, USA* (J. Peckham, ed.), pp. 255–264, ACM Press, May 1997.

[22] M. J. Zaki, "Scalable algorithms for association mining," *Knowledge and Data Engineering*, vol. 12, no. 2, pp. 372–390, 2000.

[23] M. Zaki, "Parallel and distributed association mining: A survey," *IEEE Concurrency*, vol. 7, no. 4, pp. 14–25, 1999.

[24] J. Hipp, U. Güntzer, and G. Nakhaeizadeh, "Algorithms for association rule mining – a general survey and comparison," *SIGKDD Explorations*, vol. 2, pp. 58–64, July 2000.

[25] U. M. Fayyad, D. Haussler, and P. E. Stolorz, "KDD for science data analysis: Issues and examples," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96). Portland, Oregon*, pp. 50–56, AAAI Press, 1996.

[26] W. A. Maniatty and M. J. Zaki, *Parallel and Distributed Processing*, vol. 1800 of *Lecture Notes in Computer Science*, ch. A Requirements Analysis for Parallel KDD Systems. Springer Berlin / Heidelberg, 358–365.

[27] R. Agrawal and J. C. Shafer, "Parallel mining of association rules: Design, implementation and experience," tech. rep., IBM Almaden Research Center, IBM Corp, Almaden Res Ctr, 650 Harry Rd, San Jose, Ca, 95120, 1996.

[28] F. Geerts, B. Goethals, and J. V. den Bussche, "A tight upper bound on the number of candidate patterns," in *Proceedings of the First IEEE International Conference on Data Mining*, 2001.

[29] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "Adaptive and resource-aware mining of frequent sets," in *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9–12 December 2002, Maebashi City, Japan*, pp. 338–345, IEEE Computer Society, 2002.

[30] S. Orlando, P. Palmerini, and R. Perego, "Dci: a hybrid algorithm for frequent set counting," Tech. Rep. TR-CS-01-9, Dip. di Informatica, Universit Ca' Foscari di Venezia, 2001.

[31] M. El-Hajj and O. R. Zaïane, "Parallel association rule mining with minimum inter-processor communication," in *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, (Washington, DC, USA), IEEE Computer Society, 2003.

[32] S. Cong, J. Han, J. Hoeflinger, and D. Padua, "A sampling-based framework for parallel data mining," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 255–265, ACM Press, 2005.

[33] V. Guralnik and G. Karypis, "Parallel tree-projection-based sequence mining algorithms," *Parallel Computing*, vol. 30, pp. 443–472, April 2004.

[34] A. Schuster and R. Wolff, "Communication-efficient distributed mining of association rules," *Data Mining and Knowledge Discovery*, vol. 8, pp. 171–196, March 2004.

[35] Z. K. Baker and V. K. Prasanna, "Efficient hardware data mining with the apriori algorithm on fpgas," in *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 3–12, IEEE Computer Society, 2005.

[36] Z. K. Baker and V. K. Prasanna, "An architecture for efficient hardware data mining using reconfigurable computing systems," in *FCCM '06: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, (Washington, DC, USA), pp. 67–75, IEEE Computer Society, 2006.

[37] B. B. Cambazoglu and C. Aykanat, "Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 1, pp. 3–16, 2007.

[38] C. Aykanat, B. B. Cambazoglu, F. Findik, and T. M. Kurç, "Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids," *J. Parallel Distrib. Comput.*, vol. 67, no. 1, pp. 77–99, 2007.

[39] T. Joachims, "Transductive learning via spectral graph partitioning," in *In ICML*, pp. 290–297, 2003.

[40] J. Wang, T. Jebara, and S. fu Chang, "Graph transduction via alternating minimization," in *Proc. 25th ICML*, 2008.

[41] U. Brandes, M. Gaertler, and D. Wagner, "Experiments on graph clustering algorithms," in *Algorithms - ESA 2003* (G. Di Battista and U. Zwick, eds.), vol. 2832 of *Lecture Notes in Computer Science*, pp. 568–579, Springer Berlin / Heidelberg, 2003.

[42] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, p. 281297, University of California Press, 1967.

[43] S. P. Lloyd, "Least square quantization in pcm," *IEEE Transactions on Information Theory*, 1982. Originally published in 1957 in Bell Telephone Laboratories Paper.

[44] E. Fix and J. Hodges, "Discriminatory analysis, nonparametric discrimination: Consistency properties," Tech. Rep. Technical Report 4, USAF School of Aviation Medicine, 1951.

[45] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, 1967.

[46] R. B. Marimont and M. B. Shapir, "Nearest neighbour searches and the curse of dimensionality," *Journal of the Institute of Mathematics and its Applications*, pp. 59–70, 1979.

[47] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, (San Francisco, CA, USA), pp. 194–205, Morgan Kaufmann Publishers Inc., 1998.

[48] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, pp. 273–321, September 2001.

[49] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer, "Space filling curves and their use in the design of geometric data structures," in *LATIN '95: Theoretical Informatics* (R. Baeza-Yates, E. Goles, and P. Poblete, eds.), vol. 911 of *Lecture Notes in Computer Science*, pp. 36–48, Springer Berlin / Heidelberg, 1995.

[50] K. Clarkson, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, ch. Nearest-neighbor searching and metric space dimensions. MIT Press, 2006.

[51] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *International Conference On Management of Data*, pp. 47–57, ACM, 1984.

[52] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[53] S. Brin, "Near neighbor search in large metric spaces," in *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, (San Francisco, CA, USA), pp. 574–584, Morgan Kaufmann Publishers Inc., 1995.

[54] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, (San Francisco, CA, USA), pp. 426–435, Morgan Kaufmann Publishers Inc., 1997.

[55] A. Faragó, T. Linder, and G. Lugosi, "Fast nearest-neighbor search in dissimilarity spaces," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, pp. 957–962, September 1993.

[56] B. Bustos and N. Morales, "On the asymptotic behavior of nearest neighbor search using pivot-based indexes," in *Proceedings of the Third International Conference on SImilarity Search and APplications*, SISAP '10, (New York, NY, USA), pp. 33–39, ACM, 2010.

[57] J. M. Kleinberg, "Two algorithms for nearest-neighbor search in high dimensions," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, (New York, NY, USA), pp. 599–608, ACM, 1997.

[58] R. Fagin, R. Kumar, and D. Sivakumar, "Efficient similarity search and classification via rank aggregation," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, (New York, NY, USA), pp. 301–312, ACM, 2003.

[59] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very*

Large Data Bases, VLDB '99, (San Francisco, CA, USA), pp. 518–529, Morgan Kaufmann Publishers Inc., 1999.

[60] N. Ailon and B. Chazelle, "Approximate nearest neighbors and the fast johnson-lindenstrauss transform," in *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, STOC '06, (New York, NY, USA), pp. 557–563, ACM, 2006.

[61] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, pp. 117–122, January 2008.

[62] E. Kushilevitz, R. Ostrovsky, and Y. Rabani, "Efficient search for approximate nearest neighbor in high dimensional spaces," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, (New York, NY, USA), pp. 614–623, ACM, 1998.

[63] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," in *ACM-SIAM Symposium on Discrete Algorithms*, pp. 573–582, 1994.

[64] S. Ilyinsky, M. Kuzmin, A. Melkov, and I. Segalovich, "An efficient method to detect duplicates of web documents with the use of inverted index," in *Proc 11 th Int World Wide Web Conference WWW2002*, 2002.

[65] A. Skubalska-Rafajlowicz, E.; Krzyzak, "Fast k-nn classification rule using metric on space-filling curves," in *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, pp. 121–125, Aug 1996.

[66] S. Liao, M. A. Lopez, and S. T. Leutenegger, "High dimensional similarity search with space filling curves," in *In Proceedings of the 17th International Conference on Data Engineering*, pp. 615–622, IEEE Computer Society, 2000.

[67] G. Mainar-Ruiz and J.-C. Perez-Cortes, "Approximate nearest neighbor search using a single space-filling curve and multiple representations of the

data points," in *Proceedings of the 18th International Conference on Pattern Recognition - Volume 02*, ICPR '06, (Washington, DC, USA), pp. 502–505, IEEE Computer Society, 2006.

[68] S. Kulkarni and R. Orlandic, "High-dimensional similarity search using data-sensitive space partitioning," in *Proc. 17th Int. Conf. On Database and Expert Systems Dexa*, 2006.

[69] P. Vaidya, "An $o(n \log n)$ algorithm for the all-nearest-neighbors problem," *Discrete & Computational Geometry*, vol. 4, pp. 101–115, 1989. 10.1007/BF02187718.

[70] P. B. Callahan and S. R. Kosaraju, "A decomposition of multi-dimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *J. ACM*, vol. 42, pp. 546–556, 1992.

[71] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Comput. Surv.*, vol. 33, pp. 273–321, September 2001.

[72] J. Lin, "Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '09, (New York, NY, USA), pp. 155–162, ACM, 2009.

[73] A. Awekar and N. F. Samatova, "Parallel all pairs similarity search," in *Proceedings of the 2010 International Conference on Information Knowledge Engineering*, (Las Vegas, Nevada, USA), July 2010.

[74] E. Plaku and L. E. Kavraki, "Distributed computation of the knn graph for large high-dimensional point sets," *J. Parallel Distrib. Comput.*, vol. 67, pp. 346–359, 2007.

[75] Alsabti, Ranka, and Singh, "An Efficient Parallel Algorithm for High Dimensional Similarity Join," in *IPPS: 11th International Parallel Processing Symposium*, IEEE Computer Society Press, 1998.

[76] G. Aparcio, I. Blanquer, and V. Hernndez, "A parallel implementation of the k nearest neighbours classifier in three levels: Threads, mpi processes

and the grid," in *High Performance Computing for Computational Science - VECPAR 2006* (M. Dayd, J. Palma, . Coutinho, E. Pacitti, and J. Lopes, eds.), vol. 4395 of *Lecture Notes in Computer Science*, pp. 225–235, Springer Berlin / Heidelberg, 2007.

[77] V. Olman, F. Mao, H. Wu, and Y. Xu, "Parallel clustering algorithm for large data sets with applications in bioinformatics," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 6, pp. 344–352, April 2009.

[78] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," *SIGMOD Rec.*, vol. 18, pp. 110–121, June 1989.

[79] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, (New York, NY, USA), pp. 495–506, ACM, 2010.

[80] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *J. ACM*, vol. 42, pp. 67–90, January 1995.

[81] J. W. H. Liu, "A graph partitioning algortihm by node separators," *ACM Transactions on Mathematical Software*, vol. 15, pp. 198–219, September 1989.

[82] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, January 1999.

[83] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *An Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley, second ed., 2003.

[84] R. J. Bayardo Jr., "Efficiently mining long patterns from databases," *SIGMOD Rec.*, vol. 27, no. 2, pp. 85–93, 1998.

[85] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri, "kdci: a multi-strategy algorithm for mining frequent sets," in *Proceedings of the Second IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, (Brighton, UK), 2004.

[86] T. Uno, M. Kiyomi, and H. Arimura, "Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," in *Proceedings of the Second IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, (Brighton, UK), 2004.

[87] A. Fiat and S. Shporer, "Aim2: Another itemset miner," in *Proceedings of the Second IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, (Brighton, UK), 2004.

[88] E. Özkural and C. Aykanat, "A space optimization for fp-growth," in *Proceedings of the Second IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, (Brighton, UK), 2004.

[89] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based sparse matrix ordering," in *Second International Workshop on Combinatorial Scientific Computing (CSC05), Toulouse, France*, June 2005.

[90] U. Catalyurek, C. Aykanat, and E. Kayaaslan, "Hypergraph partitioning-based fill-reducing ordering," tech. rep., Bilkent University Institute of Science and Engineering, 2009. BU-CE-0904 (Submitted to SIAM Journal on Scientific Computing. Revised version under review.).

[91] Ü. V. Çatalyürek and C. Aykanat, "Patoh: A multilevel hypergraph partitioning tool, version 3.0," tech. rep., Bilkent University, Computer Engineering Department, 1999.

[92] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, 1999.

[93] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, "Webdocs: a real-life huge transactional dataset," in *Proceedings of the Second IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, (Brighton, UK), 2004.

[94] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer, "BEOWULF: A parallel workstation for scientific computation," in *Proceedings of the 24th International Conference on Parallel Processing*, (Oconomowoc, WI), pp. I:11–14, 1995.

[95] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke, "Webbase: a repository of web pages," *Comput. Netw.*, vol. 33, pp. 277–293, June 2000.

[96] M. Brito, E. Chvez, A. Quiroz, and J. Yukich, "Connectivity of the mutual k-nearest-neighbor graph in clustering and outlier detection," *Statistics & Probability Letters*, vol. 35, no. 1, pp. 33 – 42, 1997.