SCALABLE MINING ON EMERGING ARCHITECTURES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Gregory Buehrer, MS

* * * * *

The Ohio State University

2008

Dissertation Committee:

Dr. Srinivasan Parthasarathy, Adviser

Dr. Paolo A.G. Sivilotti

Dr. P. Sadayappan

Approved by

Adviser

Graduate Program in

Computer Science and

Engineering

ABSTRACT

Recent advances in data collection technology have generated large-scale data stores. Examples include the Internet, scientific simulation results, and government identification databases. Current estimates of the size of the Internet, which is a loosely structured public database, are at 20 billion pages and 250 billion links. Data mining is the process of discovering interesting and previously unknown information from such stores. Techniques effective on small data sets simply do not scale to larger data sets. Thus, as the ability to collect and store data increases, our ability to make use of the data degrades. This degradation is due to two main issues. First, the utility of efficient serial algorithms to mine such data is often lowered because this data is distributed on multiple machines. Second, since the complexity of most mining algorithms is polynomial or even exponential with the input size, even on parallel machines the runtimes exceed practical limitations. This dissertation addresses the concerns of mining large data stores by providing improvements to the state of the art in two directions. First, mining algorithms are restructured to glean the benefits of emerging commodity hardware. Second, we identify a set of useful patterns, and formulate algorithms for extracting them in log-linear time, enabling scalable performance on large datasets.

We make several contributions towards data mining on emerging commodity hardware. First, we leverage the parallel disks, memory bandwidth and large number of

processors to mine for exact global patterns in terascale distributed data sets. We provide a 10-fold improvement to the existing state of the art in distributed mining. Second, we leverage the improved computational throughput of emerging CMPs to provide nearly-linear scale-ups for shared-memory parallel structure mining. Third, we explore data mining on the Cell processor, re-architecting clustering, classification and outlier detection to glean significant runtime improvements. These improvements are afforded by the high floating point throughput of this emerging processor. We also show examples where the Cell processor requires more compute time than competing technologies, primarily due to its high latency when exchanging small chunks from main memory.

We identify an important class of itemset patterns that can be extracted in log-linear time, improving significantly on the scalability afforded typically by frequent itemset mining algorithms. The algorithm proceeds by first hashing the global data set into partitions of highly similar items, and then mining the localized sets with a heuristic, single projection. We then show how this technique can be used to compress large graphs, improving on the state of the art in web compression. Finally, we leverage the techniques developed to build a general-purpose placement service for distributed data mining centers. The placement has low complexity and affords highly scalable solutions to many common data mining queries.

ACKNOWLEDGMENTS

My research has been supported in part by grants from the National Science Foundation (#NGS-CNS-0406386, #CAREER-IIS-0347662 and #RI-CNS-0403342), the Ohio Department of Transportation, and a Microsoft Research Fellowship from Live Labs. Any opinions, findings, and conclusions or recommendations expressed here are those of the author and, if applicable, his adviser and collaborators, and do not necessarily reflect the views of these funding sources.

I would like to thank Dr. Srinivasan Parthasarathy (Srini) for his tireless efforts towards my academic maturation. Through my badgering, complaining, flip-flopping, the additional constraints imposed by my family–he was wonderfully patient throughout. His gave words of encouragement with the paper rejections, handshakes on the acceptances. He always puts the student first, and for that I am forever thankful. He made my strengths shine and cleaned up my weaknesses.

I also would like to thank Dr. Paolo A.G. Sivilotti for his patience, direction and support. His five years of advise proved invaluable. I couldn't count the times I stuck my head inside his door and interrupted his train of thought, if only to discuss the ramifications of a run-on-first-and-second-down offense.

I would like to thank Dr. P. Sadayappan, whose 888 reading groups and profoundly inquisitive nature helped shape my understanding of performance computing.

To my labmates in the DMRL and those in the SE lab–a special thanks for working through the papers, rejections, 888 sessions, practice talks (with slide by slide mind-numbing reviews), and Oxleys lunches that make up life in the CSE department. I look forward to seeing them all out in the wild.

To my children, Viviana and Oliver, who patiently waited up in bed for me to come home so many nights. You both came to be while I attended graduate school, sacrificed along with me, and motivated me to finish.

Most importantly, to my wife Jennifer, whose dedication, effort and sacrifice dwarfed my own. Without you I truly would not have made it. I can't possibly enumerate the days I wished I could have done more at home, the days you did my work there for me. It was a journey we shared, a path we took in a time that often required more than I had. You have made the difficult times bearable and the good times great.

VITA

October 15th, 1971 ..... Born - Toledo, OH

1994 ..... B.S. Chemical Engineering

2006 ..... M.S. Computer Science

PUBLICATIONS

Research Publications

G. Buehrer and S. Parthasarathy "A Placement Service for Data Mining Centers". *OSU-CISRC-11/07-TR75*, (in submission).

G. Buehrer, K. Chellapilla and S. Parthasarathy "Itemset Mining in Log-linear Time". *OSU TR*, (in submission).

G. Buehrer and S. Parthasarathy "The Potential of the Cell Broadband Engine for Data Mining". *OSU-CISRC-3/07-TR22*, (in submission).

G. Buehrer and K. Chellapilla "A Scalable Pattern Mining Approach to Web Graph Compression with Communities". *To appear in the Proceedings of the International Conference on Web Search and Data Mining (WSDM)*, Feb. 2008.

G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz "Toward Terabyte Pattern Mining: An Architecture-conscious Solution". *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), March 2007*

A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.Chen, and P. Dubey "Cache-Conscious Frequent Pattern Mining on Modern and Emerging Processors". *International Journal on Very Large Data Bases (VLDBJ), 2007*

A. Ghoting, G. Buehrer, M. Goyder, S. Tatikonda, X. Zhang, S. Parthasarathy, T. Kurc and J. Saltz "Knowledge and Cache Conscious Algorithm Design and Systems Support for Data Mining Algorithms". *Proceedings of IPDPS NGS Workshop (IPDPS NGS), April 2006*

G. Buehrer, S. Parthasarathy, A. Ghoting, D. Kim, A. Nguyen, Y.Chen, and P. Dubey "Efficient Frequent Pattern Mining on Shared Memory Systems: Implications for Chip Multiprocessor Architectures". *Proceedings of the ACM SIGPLAN Memory Systems Performance and Correctness Workshop at ASPLOS (MSPC), 2006*

G. Buehrer, S. Parthasarathy, and Y. Chen "Adaptive Parallel Graph Mining for CMP Architectures". *Proceedings of the IEEE International Conference on Data Mining (ICDM), 2006*

G. Buehrer, S. Parthasarathy, and A. Ghoting "Out-of-core Frequent Pattern Mining on a Commodity PC". *Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD), 2006*

G. Buehrer, S. Parthasarathy, D. Kim, Y.Chen, A. Nguyen, and P. Dubey "Towards Data Mining on Emerging Architectures". *Proceedings of 9th SIAM High Performance Data Mining Workshop (HPDM), April 2006*

G. Buehrer, A. Ghoting, X. Zhang, S. Tatikonda, S. Parthasarathy, T. Kurc, and J. Saltz "I/O Conscious Algorithm Design and Systems Support for Data Analysis on Emerging Architectures". *Proceedings of IPDPS NGS Workshop (IPDPS NGS), April 2006*

G. Buehrer, B. Weide and P. Sivilotti "Using Parse Tree Validation to Prevent SQL Injection Attacks". *Proceedings of the Fifth FSE International Workshop on Software Engineering and Middleware (FSE SEM), September 2005*

A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, Y.Chen, A. Nguyen, and P. Dubey "Cache-Conscious Frequent Pattern Mining on a Modern Processor". *Proceedings of the 31st International Conference on Very Large Databases (VLDB), September*

A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey "A Characterization of Data Mining Workloads on a Modern Processor". *Proceedings of the 2005 SIGMOD Data Management on New Hardware Workshop (DAMON), June 2005*

H. Wang, S. Parthasarathy, A. Ghoting, S. Tatikonda, G. Buehrer, T. Kurc, and J. Saltz "Design of a Next Generation Sampling Service for Large Scale Data Analysis Applications". *Proceedings of the International Conference on Supercomputing (ICS), June 2005*

H. Wang, A. Ghoting, G. Buehrer, S. Tatikonda, S. Parthasarathy, T. Kurc, and J. Saltz "A Services Oriented Framework for Next Generation Data Analysis Centers". *Proceedings of the IPDPS NGS Workshop (IPDPS NGS), April 2005*

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in Scalable Data Mining: Prof. Srinivasan Parthasarathy

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

At the heart of any advanced civilization is the ability to collect and use information. For example, consider the Great Roman Empire. A key to their rise and dominance of Europe for almost 400 years was their ability to improve the state-of-the-art in architecture, such as improved irrigation via concrete aqueducts. Improved irrigation afforded improved agriculture, as well as the ability to settle previously unusable lands. Interestingly, this was largely fueled from their discovery of hydraulic cement, produced from a mixture of lime, stone and volcanic ash. Hydraulic cement's novelty is that it will solidify even when submerged. The Romans taught the technique to each new conquered territory, importing ash from the region immediately surrounding Mount Vesuvius. Unfortunately, with the fall of the Roman Empire this information was lost. It would be almost 1000 years before this knowledge was rediscovered.

Today, we can collect and record data rather effectively, and the possibility of losing such an important formula is unlikely. In fact, we store quite a bit of data. Over the past decade, technological advances have allowed us to gather an increasing amount of data in the areas of science, engineering, and business. We typically define

data to be transactions or records in a database, although it may also pertain to text documents, binary files, or even multimedia files.

To address the challenge of extracting actionable information from these large data sources, organizations and researchers have increasingly turned to data mining. Data mining resides at the crossroads of several fields of study; statistics, high performance computing, machine learning, and database systems, among others. As an example of actionable information, suppose a pharmaceutical company were to examine the molecular structure of every known carcinogen. They may pose several basic queries, which traditional database technology cannot answer. *Is there a pattern to these molecules? If so, what is it? Could a recurring substructure provide insight into other molecules which may cause cancer?*

Unfortunately, our ability to collect this data far outstrips our ability to analyze it in a timely manner. Efficiency is critical to the KDD process, since it is inherently iterative (repetitive), involves a human-in-the-loop (interactive), and without optimization it can require many days to process even a single query. Many existing algorithms, both parallel and sequential, are extremely slow when faced with truly large datasets. In this dissertation, we target two primary reasons for poor efficiency in data mining applications. First, many data mining algorithms are computationally complex and scale polynomially or worse with the size of the input data set. Second, most data mining solutions largely under-use the available hardware. Much of the research to date in data mining has focused on serial algorithms that intelligently prune an exponential search space. These improvements do not address either a) the inherent computional complexity of the underlying challenge, or b) the algorithm's ability to leverage emerging parallel technology.

1.1 Addressing the Need to Leverage Emerging Commodity Parallel Hardware

As existing hardware goes underused in data mining applications, computer manufacturers continue to revolutionize the PC. Of particular interest are emerging parallel commodity CPUs. These innovations include Chip Multiprocessors (CMPs) and simultaneous multithreading (SMT). CMPs provide true multiple-instruction, multiple-data (MIMD) parallel computing in an inexpensive, commodity package by incorporating multiple processing elements on the same die. These cores are independent computational units which may or may not share level two cache. They can execute multiple threads concurrently because each processing element has its own context and functional units. As these low cost parallel chips become mainstream, designing data mining algorithms to leverage them becomes an important task. Key challenges to overcome are a) a lack of sufficient task-level granularity, b) less main memory per processor than traditional shared-memory multiprocessors, and b) the limited bandwidth of the memory bus. To leverage these advancements, algorithm designers must develop strategies to maintain high throughput by generating a task creation model that can provide sufficient granularity without requiring significant overheads. Although there are many parallel algorithms in the data mining literature, almost none are capable of mining tera-scale data, and providing the granularity needed to be effective on CMP systems.

Second, emerging commodity message-passing clusters are improving by leaps and bounds. They afford exceptional computional power, large disk space, large aggregate main memory, and efficient interconnects. These four features offer the opportunity to mine very large data sets. However, most data mining research explores mining

rather small data sets, even when using distributed clusters. This is in part due to the fact that working with large data sets is quite challenging. To begin with, data mining algorithms often suffer from a significant amount of load imbalance, since it is difficult to predict the execution time for a specific task. Consequently, extant parallelization strategies fail to use all the resources available on modern-day clusters. Furthermore, different tasks in data mining often need to share state or exchange information. For small data sets, this constant communication does not appear to be a concern. However, when mining tera-scale data it is costly, particularly when the algorithms store their meta data in dynamic, pointer-based structures. Serializing and transferring this data quickly becomes the bottleneck to performance. Surprisingly, a small percentage of the data mining literature addresses attempts to mine terabyte data sets using parallel, chip-level hardware. However, it is clear that a) future PCs will use CMPs, b) CMPs systems will be the building blocks of next generation clusters, and c) many emerging applications have extremely large data sets.

1.2 Addressing the Need to Reduce the Computational Complexity

Most all data mining algorithms scale polynomially (or worse) with some properties of the input data. For example, clustering using k-nearest neighbors is $O(n^2)$, frequent pattern mining using *Apriori*-type algorithms is $O(2^L)$, and outlier detection using *ORCA* is $O(n^2)$. However, to mine truly large data, such as the data sets drawn from the web domain, in many cases even $O(n^2)$ is cost-prohibitive. For example, conservative estimates place the number of web pages to be over 15 billion. If we were to perform an $O(n^2)$ comparison algorithm with a cost of only one cycle per comparison, a 3GHz machine will require over 2000 years to finish. In many cases, researchers

have developed approximate algorithms that have reduced the complexities of data mining algorithms. However, this is not the case for pattern mining.

Pattern mining is a fundamental class of data mining algorithms that seeks to discover interesting, recurring patterns in data. Unfortunately, algorithms which produce an exact solution scale exponentially with the number of different input labels in the data. Existing research addresses this challenge by finding clever ways to reduce the number of patterns that must be checked [3, 117, 52], but in all cases, the complexity of the problem remains unchanged. In this dissertation, we will formulate algorithms that limit the computational complexity of itemset mining to log-linear, while still affording many of the useful results achieved by traditional algorithms.

1.3 Thesis Statement

To enhance the scalability of data mining algorithms, researchers must understand and leverage the recent advantages provided by emerging commodity hardware. Also, pattern-mining algorithm designers should develop techniques to glean relevant and useful patterns while maintaining log-linear computional complexities. By abiding by these two guidelines, algorithms can be designed that are capable of scaling to truly massive data sets.

1.4 Strategies for Scaling to Massive Data Sets

Before we[1] discuss our contributions, we present several strategies for developing data mining algorithms that are capable of mining massive data sets efficiently.

[1]The word *we* is used as a personal preference. The primary contributor on the work presented in this dissertation is the author.

- We must break our habit of developing serial solutions. In developing these parallel algorithms, we must consider a fine-grained partitioning mechanism to accommodate a large number (possibly thousands) of compute cores. This partitioning should adapt to the available resources at run time as needed, and use a distributed queuing model. These techniques will offer proper load balancing.

- We must explicitly consider the amount of memory we consume. Data mining algorithms often use meta data to avoid costly data set scans. However, large data sets invariable grow meta structures. We must develop algorithms that leverage the available addressable memory.

- We must leverage specialized hardware, if available. For example, many future processor designs will incorporate specialized cores designed to increase floating-point throughputs. Algorithm designers should consider multiple hierarchies in parallel design so as to leverage these specialized cores.

- We must develop approximate solutions for workloads when possible. These approximate solutions should be limited to log-linear computional complexities, so as to offer extremely scalable solutions when an exact solution is not needed. Log-linear is a modest increase over a linear complexity, but it affords sorting.

1.5 Contributions

We follow our guidelines to make the following contributions. First, we develop a parallel approach to global itemset mining for terascale data sets on a message-passing distributed cluster of machines. Our solution relies on an algorithmic design

strategy that is memory-, disk- and network- conscious. Our approach minimizes synchronization between nodes, while also reducing the data set size required at each node. The sequential algorithm makes use of tree structures to efficiently search and prune the space of itemsets. We devise a mechanism for efficient serialization and merging of local trees called *strip-marshaling and merging*, to allow each node to make global decisions as to which itemsets are frequent. In addition, local tree pruning reduces the data structure at each node by a factor of up to 15-fold on 48 nodes. Moreover, this reduction increases with increasing cluster size. Our overall execution times are more than 10 times faster than existing solutions[19], such as *CD, DD* [2], *IDD* and *HD* [50].

We also develop a mechanism, called adaptive partitioning, to dynamically partition tasks onto compute nodes for CMP systems. In many pattern-mining settings, extant task creation mechanisms do not generate enough tasks for proper load balancing. This is in large part because the time to mine a task is often not known *a priori*. By varying the size of a task at runtime, dynamic partitioning allows work to be evenly divided among the available processing nodes. We illustrate the efficacy of the technique in the context of parallel frequent substructure mining. The algorithm also employs dynamic memory meta structures when memory constraints and task dependency relationships afford it. Scalability improves from 5-fold to over 20-fold on 32 processing nodes on several well-cited, real world data sets. We present these techniques in the context of CMP and SMT architectures, where thread-level parallelism is a primary concern towards leveraging the underlying hardware.

We further our study of CMPs by investigating the ability of the Cell processor [48] to mine key data mining kernels. The Cell is a new CMP with similarities to

both POSIX-style parallel processors and graphic processors. We develop efficient clustering, classification and outlier detection algorithms, and illustrate the potential of the Cell processor for streaming computations. These are three fundamental data mining tasks, often used independently or as precursors to solve multi-step data mining challenges. We pinpoint the bottlenecks in scalability and performance when mapping these workloads to this platform, resulting in solutions that are up to 50 times faster than similar solutions executed on competing technologies.

Next, we relax the definition of pattern mining (presented in Chapter 2), such that the exact support of an itemset is not necessary; rather a lower bound is sufficient. Then, a highly scalable solution is presented. The algorithm first localizes the input data into similar subsets so that locally interesting patterns can be found at low cost. It has the benefit of finding long patterns with high utility, without enumerating first all sub patterns, or traversing the search space in a particular order. The algorithm has a log-linear computional complexity, is simple to implement, and parallelizes easily. We evaluate it against other itemset mining strategies, namely closed itemsets. Lastly, we present a simple new metric for itemset utility. It incorporates both the frequency of a pattern and its length to rank it against other patterns.

We build on our log-linear approximate itemset miner to develop a compression scheme for very large graphs. We apply the technique to the web graph. Specifically, we formulate a link server compression and community discovery mechanism with the following properties. First, the compression ratio is high, often higher than the best known compression techniques in the literature. Community discovery queries can be performed quickly, often finding seed patterns in constant time. The compression supports random access to the web graph without decompressing. Lastly, it does not

require sorted and adjacently labeled vertices, such as by URL, and thus supports incremental updates.

Our final contribution is a data-placement service for next-generation data centers for cloud computing. We leverage the technologies previously described to help define an API for partitioning data onto the machines in the cluster to facilitate accelerated mining. In some cases, it also reduces computation times. For example, the placement service can lower the number of inter-machine communications when walking large graphs by co-locating similar graph nodes on the same machine. The service also uses manifests to manage disjoint partitions of the data, in an effort to lower transfer times between computations.

1.6 Organization

The rest of this document is organized as follows. Chapter 2 presents the relevant existing research regarding data mining for very large data sets. Chapter 3 offers our solution for mining very large transactional (retail) data sets for exact global itemsets. Chapter 4 develops strategies for mining patterns on emerging chip multiprocessors, and presents our detailed solution for mining complex structures in graphical data sets. Chapter 5 explores the tradeoffs of the Cell processor for performing several data mining workloads. Chapter 6 describes our approximate itemset mining technique, which bounds the computational complexity to log-linear time. We include a multi-level parallelization for a cluster of distributed CMPs. In Chapter 7, we leverage this approximate mining technique to find communities in web graphs, and subsequently use these communities to compress the web. Chapter 8 leverages the technologies presented in the previous chapters to construct a distributed placement service for

data mining clusters. We present concluding remarks and the direction of future work in Chapter 9.

CHAPTER 2

RELATED RESEARCH

In this chapter, we discuss the relevant research on the scalability of data mining applications. Much of the work presented in later chapters addresses algorithms for mining recurring patterns in data. Therefore, before presenting the related research, a definition of frequent itemset mining is provided. Subsequent chapters will alter this definition slightly. These alterations will be in reference to the following definition, and will be defined at that time.

The frequent pattern mining problem was formulated by Agrawal *et al.* [1] for association rule mining. Briefly, the problem description is as follows: Let $I = \{i_1, i_2, \cdots, i_m\}$ be a set of M items[2], and let $D = \{T_1, T_2, \cdots, T_n\}$ be a set of N transactions, where each transaction T_i is a subset of I. Let $|D|$ be the total number of items in D, or $|D| = \sum_{j=1}^{N} |T_j|$. For simplicity, when will use D as a scaler, it will represent $|D|$. The *support* of i is $\sum_{j=1}^{m}(1 : i \subseteq T_j)$, or informally speaking, the number of transactions in D that have i as a subset. The frequent pattern mining problem is to find all $i \in I$ with *support* greater than or equal to the minimum support σ. An association rules is an implication in the form $X \rightarrow Y$, where X and Y are

[2]Sometimes the set I is called the label set.

frequent disjoint sets. For example, *Verve* → *Interpol*, i.e. *if a customer purchases a Verve CD, she is likely to purchase an Interpol CD too.*

2.1 Serial Frequent Itemset Mining

Agrawal and Srikant formulated *Apriori* [3], the first efficient algorithm to solve itemset mining. *Apriori* traverses the itemset search space in breadth-first order. Its efficiency stems from its use of the anti-monotone property: *If a size k-itemset is not frequent, then any size $(k+1)$-itemset containing it will not be frequent.* The algorithm finds all frequent 1-items in the data set, and then iteratively finds all frequent l-itemsets using the frequent $(l-1)$-itemsets discovered previously.

This general level-wise algorithm has been extended in several different forms leading to improvements such as *DHP* [80] and *DIC* [14]. *DHP* uses hashing to reduce the number of candidate itemsets that must be considered during each data set scan, particularly by precomputing frequent-2 items during the initial data set scan. Furthermore, it progressively prunes the transaction data set as it discovers items that will not be useful during the next data set scan. *DIC* processes the data set in chunks and considers new candidate itemsets as they are discovered in the chunk. Unlike *Apriori*, *DIC* does not require that the entire data set be scanned for each new candidate. Such an approach affords fewer passes.

Zaki, Parthasarathy, Ogihara and Li proposed *Eclat* [117], and several other algorithms that use equivalence classes to partition the problem into independent subtasks. The use of the vertical data format in *Eclat* allows for fast support counting by set intersection operations. The independent nature of subtasks, coupled with the use

of the vertical data format, results in improved I/O efficiency because each subtask is able to reuse data in main memory.

Han, Pei and Yin developed *FPGrowth* [52], an algorithm that significantly reduces the number of data set scans required. *FPGrowth* summarizes the data set into a succinct prefix tree or *FP-tree*. An additional benefit is that it does not have an explicit candidate generation phase. Rather, it generates frequent itemsets using tree projections in main memory. The payoff is an improved search space traversal. However, the pointer-based nature of the *FP-tree* requires costly dereferences.

Liu *et al.* [71], developed *AFOPT* to mine for itemsets. *AFOPT* reverses the prefix tree employed by *FPGrowth*, allowing the least frequent item to be mined first. The benefit is that once the least frequent item has been mined, the portion of the prefix tree containing it can be discarded. This results in a much smaller memory footprint than that of *FPGrowth*. However, it hinders parallel performance. Also, even *AFOPT* will exceed the available memory of a commodity PC when the data set generates sufficient meta data. As an example, mining the webdocs data set at a 5% support on a 1GB RAM PC will exceed main memory. Once the pointer-based tree exceeds main memory, its lack of locality causes a large number of page faults, and results in poor execution times. Therefore, it alone does not provide out-of-core scalability.

Toivonen [99], showed that sampling large databases produced sufficient information to mine for association rules. The algorithm roughly proceeds as follows. In step one, a small sample is mined for frequent itemsets. In step two, the results are used to create a negative border, or the set of infrequent itemsets one link from the frequent itemsets in the search lattice. Next, the database is scanned again, and the

frequent set and negative border are counted. If any itemset on the negative border is frequent, or any locally frequent itemset is not frequent, the set used for counting is adjusted accordingly and another scan is made. When a scan produces no failures, the algorithm terminates. A drawback of this approach is that there is no bound on the number of required scans. Also, as the support is lowered, the size of the sample required to accurately estimate the final result set increases dramatically.

Several researchers have worked to lower the number of frequent patterns produced by the mining algorithms, resulting in definitions for *closed* itemsets and *maximal* itemsets. A closed itemset [116] is a frequent itemset that is not a subset of another frequent itemset with the same support. In this way, closed patterns do not discard information. Zaki proposed *CHARM* to mine for closed itemsets. It enumerates closed sets using a dual itemset-tidset search tree. It also employs diffsets to reduce the memory footprint of intermediate computations.

Maximal patterns [45] are similar to closed patterns, but instead of omitting the output of all proper subsets with the same support, *all* subset patterns of frequent set patterns are discarded, regardless of support. Therefore, some information is lost, namely the exact support of the frequent subset patterns. Burdick, Calimlim and Gehrke proposed MAFIA [21], to enumerate frequent maximal itemsets. *MAFIA* is a depth-first algorithm that embeds the maximal property into the mining process as a constraint, to effectively traverse the search lattice. It combines a vertical bitmap representation of the database with an efficient compression schema. The use of bitmaps provides an order of magnitude improvement over previous techniques.

Jaroszewics and Simovici used artificial intelligence to discover the most interesting subset of the full set of frequent patterns [57]. They employed a Bayesian network to discover itemsets with frequencies that are highly unlikely based on their subsets.

Non-derivable itemsets [22] represent all itemsets for which the support cannot be derived by other reported sets. *Profile Patterns* [113] summarize the result sets by finding patterns that represent the majority of the output. In all four cases – closed patterns, maximal patterns, non-derivable patterns, and profile patterns – the support must be global and the complexity remains exponential.

There exists work to enumerate the *Top-K* patterns. These algorithms are designed to use a user-specified constraint to order the result set, and return the top K of them based on a particular sort order. The first such work was done by Webb [105] for an algorithm called *OPUS*. K-optimal constraint mining is effective for many applications, however as noted by the authors, the worst case complexity remains exponential. The target data sets presented in Chapter 6 represent this worst-case scenario, because i) it has billions of labels in the search space, ii) the desired minimum support is very low support (support=2), and iii) the distribution follows a power law, where some transactions are millions of items long. Webb has also explored post-itemset rule pruning [104]. In our work, we are only concerned with the itemsets, and do not subsequently generate rules.

TFP [103] by Wang, Han, Lu and Tzvetkov finds closed patterns of a minimum specified length, and can be thought of as a special case of the previous work, with a specialized compute algorithm. It uses the support of the itemset to order the closed result set, reporting the first K itemsets with at least *min_length* items. The user

must still provide a threshold, namely the minimum length, the algorithm has worst case exponential computation time, and uses support as the interestingness criteria.

Seno and Karypis developed *LPMiner* [94] to find all itemsets with a constant support constraint. The algorithm seeks to find patterns whose support decreases with increasing length. It is built atop FPGrowth [52], using three pruning techniques to reduce the search space. This algorithm has worst case exponential computational complexity and still relies on the notion of support to guide the search space traversal. Thus, it does not scale to our target datasets.

Savasere *et al.* presented *Partition* [91], an approach for out-of-core itemset mining. They propose to subdivide the original data set into smaller data sets that can be processed in main memory. Then, each of these partitions is mined in main memory using an in-core algorithm. This is followed by a union operation on all locally frequent itemsets discovered in each partition. Finally, to find the exact set of frequent itemsets, exact counts of all itemsets in the union are determined using a data set scan.

Grahne and Zhu presented *Diskmine* [46], an approach that uses projections to partition the data. The approach recursively projects the data set until it fits in main memory and then uses an in-core algorithm to mine at that point. This technique affords improved memory system usage at the expense of a larger search space and increased computation. They use the projection mechanism developed for *FP-Growth*. When the first projected tree fits in main memory, the technique is serviceable. However, when this is not the case, excessive data set scans slow the algorithm considerably.

Parthasarathy has shown that sampling can be effective when mining large data sets [81]. By leveraging a progressive sampling technique, the mining processed can be reduced significantly. This is in agreement with the techniques described earlier in Toivonen's research [99]. The efficiency of the method degrades when minimum support is low, as runtimes increase exponentially.

2.1.1 Streaming Algorithms

Several streaming algorithms have been proposed to mine for frequent itemsets [73, 62]. These works target very large data sets, even infinite streams. The streaming environment is a resource-constrained problem. In some cases, such as network packet processing, both the computational complexity and the space complexity are of concern. The primary goal of itemset stream miners targets the space constraint, and does not attempt to circumvent the exponential computational complexity. Often times, a sliding window is used. This window can be considered a locality in a similar manner as the min hashing schemes; however, the locality is strictly based on chronology.

For example, Jin and Agrawal mine streaming data [62] using a one-pass algorithm. They extend the streaming counting work by Karp, Papadimitrious and Shanker for streaming singletons, to mine for approximately frequent itemsets. Further, they show that exact itemsets can be discovered by making an additional pruning pass. The strategy essentially mines the complete lattice in a single pass, trimming items that can be shown to be infrequent along the way. The authors show that it does not consume a large amount of main memory for several real and synthetic data sets. However, as it finds all frequent itemsets, its runtime is required to be

exponential with the input data set, and must hold a superset of the full result set in memory at one time.

2.2 Mining Complex Patterns

The notion of a frequent pattern has been extended beyond transactional data. Sequence mining was first proposed by Agrawal and Srikant [4]. Essentially, an order is placed on the items in a transaction, and then the data is explored for recurring patterns. Generally, sequences need not be connected. Zaki *et al.* proposed the serial algorithm SPADE [114], which resulted in improved execution times. Parthasarathy, *et al.* developed incremental methods to mine for sequences [85].

2.2.1 Frequent Graph Mining

Advantages for representing data as graphs have also been widely studied. *FSG* [70], proposed by Kuramochi and Karypis, was the first graph miner to enumerate all frequent connected subgraphs. It employs an *Apriori*-like breadth-first search, creating new fragments by joining smaller ones that share a common substructure. A canonical label is generated from an adjacency matrix, to avoid duplicate computation. FSG does not maintain detailed mappings between the currently mined graph and its position in the data set (called embeddings). Instead, it uses a TID list with each frequent subgraph, and joins the list with other TID lists when it extends the graph. This is in much the same manner as *Eclat* [117].

Wang and Parthasarathy [102], developed the *Motif Miner* toolkit for mining molecular databases of recurring patterns. *Motif Miner* searches for interesting substructures in noisy geometric graphs (or graphs embedded in a three dimensional

space) targeted at large biochemical molecules, such as proteins. They furthered this work with a parallel algorithm [82] for shared memory platforms.

Yan and Han developed *gSpan* [110, 111], which solves the same problem statement as *FSG*. It traverses the search space in depth-first order, using a canonical representation called a DFScode, which is essentially the lexicographical output from a depth-first walk of a graph. Since graphs have no root, there may be many valid DFScodes for a single graph. The lexicographically largest of these encodings is considered the canonical label for that graph. Every prefix of a canonical label is also canonical. *gSpan* restricts pattern growth to the right-most path, since all extensions to any other path will result in graphs with labels which are not canonical. *gSpan* does not maintain mappings between the currently mined graph and its location in the data set. This allows for an implementation which uses very little memory. Wörlein *et al.* [108] extend *gSpan* to mine for embedded subgraphs in serial; their program is named *Edgar*. A study of *gSpan* is provided in Chapter 4.

Gaston [78] was developed by Nissan and Kok, and is similar to *gSpan* in several aspects. To begin with, it is also a depth-first traversal of the search space. In addition, *Gaston* employs a canonical representation to reduce the mining workload. However, unlike *gSpan* which uses the same labeling mechanism for each graph, *Gaston* uses a different mechanism for paths, trees and graphs. This work illustrates that by leveraging some knowledge about the structure, the number of canonical representation validations can be lowered. In particular, for trees *Gaston* can evaluate whether an extension will result in a new canonical representation in constant time. *GastonEL* introduces a pointer-based embedding structure to maintain the locations

of the currently mined graph (whereas *GastonRE* does not). Therefore, the computation required to extend a pattern is reduced. A study of *Gaston* is presented in Chapter 4.

Huan, Wang and Prins developed *FFSM* [55] to mine for frequent subgraphs. It uses a vertical search scheme within a proposed algebraic graphical framework to traverse the search lattice. It employs a canonical labeling using the concatenation of the columns in an adjacency matrix, similar to that proposed by *FSG*. *FFSM* can prune the search space based on the last column in this matrix, an ability not shared by *FSG*. However, *FFSM*'s meta data consumes excessive main memory, and it does not provide a execution time performance over previous algorithms.

Cook *et al.* [33, 54], have been developing a system called Subdue for several years. Their research has produced a parallel implementation, which solves a somewhat different problem, using approximations and user interaction to return potentially interesting substructures again on geometric graphs.

MoFa [13] is a depth-first graph miner proposed by Borgelt and Berthold. It performs a depth-first search, creating new potentially frequent subgraphs by extending smaller ones. It maintains similar embeddings to those proposed by *Gaston*. Unfortunately, the embeddings consume significant memory, particularly for small graphs which exist in many data set graphs. *MoFa* has been given much attention in the graph mining community, in part due to its flexibility. Hofer, Borgelt and Berthold [53] extend it to to mine large data sets for frequent graphs with wildcards. Fatta and Berthold [37] develop methods to dynamically load balance distributed graph mining with *MoFa*. Meinl *et al.* [77] parallelize it for shared-memory systems. Their implementation *ParMol* [76], achieves a 7-fold speedup on 12 nodes. They provide this

code and several serial graph mining implementations in Java for public download. *ParMol*'s single node execution times are far slower than what we present in Chapter 3, often by two orders of magnitude, and it also uses much more memory. This is because the underlying algorithm is slower than *gSpan* [109]. In addition, *MoFa* inherently suffers scalability challenges because it has no canonical representation, which results in performing some tasks multiple times when operating in parallel.

Yan, Yu and Han developed a graph indexing system called *gIndex* [112] to improve query execution times on graph data sets. They leverage existing work on frequent subgraph patterns to develop an index supporting the frequent pattern search. As opposed to indexing every fragment, they choose discriminatory fragments to reduce the space maintained. However, it is not clear that the searching process using this index can be parallelized to leverage multiple compute cores.

2.3 Parallel Shared-memory Data Mining

Strategies for mining associations in parallel have been proposed by various researchers.

There have been several approaches for parallel mining of other frequent patterns [86, 120]. Zaki [115], proposed parallel partitioning schemes for sequence mining. The author illustrates the benefits of dynamic load balancing in shared memory environments. However, a key difference is that estimates for task execution time (used by the author) are far easier to compute in sequence mining than for graph mining. Guralnik and Karypis had similar findings [49].

Jin and Agrawal has several works targeted at solving parallel data mining workloads [60, 61]. They implement a framework for fast prototyping of data mining workloads on shared memory systems.

There is also research by the community which addresses shared-memory data mining for problem definitions outside pattern mining. Shafer, Agrawal and Mehta [95] develop parallel classification techniques for large data sets. Joshi, Karypis and Kumar [63] extend this work. Andrade, Kurc, Saltz and Sussman [6] developed a parallel approach for decision tree construction for clusters of shared-memory machines. Zaki, Ho and Agrawal [115] developed a parallel classification algorithm for shared memory multiprocessors.

2.4 Parallel Distributed Data Mining

Many researchers have leveraged clusters of workstations [26] to mine for association rules. Zaki provided a survey of the work [120]. Agrawal and Shafer were the first to address the challenge of frequent itemset mining on shared-nothing architectures, proposing Count Distribution (*CD*) and Data Distribution *DD*[2]. Both *DD* and *CD* are based on *Apriori*. As stated prior, *Apriori* algorithms traverse the itemset lattice in a breadth-first manner. At level k, candidate itemsets of size k are generated by using frequent itemsets of size $k-1$. These candidates are then validated against the database (usually by a full scan) to obtain k-size frequent itemsets. *Apriori*-based algorithms speed up the computation by using the *anti-monotone* property and by keeping some auxiliary state information. *CD* partitions the data evenly among nodes in the cluster. The candidate support counting phase is parallelized, leveraging the

parallel I/O of multiple disks. Then, each node communicates its counts to all other nodes. Each node calculates the next round's candidates independently.

DD is the complement of *CD*. At the start of each level, each node takes a turn broadcasting its local data set to every other node. Each machine generates a mutually disjoint partition of candidates. At each level, frequent itemsets are communicated so that each processor can asynchronously generate its share of candidates for the next level. *DD* overcomes the bottleneck of serial candidate generation by partitioning generation amongst the processors. However, it incurs very high communication overhead when broadcasting the entire data set, which is especially costly in case of large datasets, where the full data set does not fit on the disk of a single machine. *CD* was shown to be far more scalable than DD. We provide a study of *CD* and *DD* in Chapter 3.

Cheung *et al.* developed *FDM* [27] as an improvement to *CD*. The main contribution of *FDM* is that it lowers the number of candidates considered for counting. Since the number of candidates is reduced, the cost of communication is lowered as well, because counts for each candidate are broadcast be each node in the cluster. *FDM* accomplishes this reduction by eliminating candidates which are known to be locally infrequent at that site. As an additional pruning step, if not all subsets of a candidate are not locally frequent, then it is pruned. The technique was modified to use less polling for a shared-memory version called *FPM* [28]. In both *FPM* and *FDM*, the number of full data set scans is proportional to the length of the longest frequent pattern.

Decision Miner [92] was proposed by Schuster and Wolff. It reduces the communication costs of *CD* by pruning candidates which cannot be frequent. The work is

similar to *FDM*, but also provides theoretical bounds for the amount of communication reduction. As does *FDM*, *Decision Miner* makes the same number of full data set scans as *Apriori*.

Han, Karypis and Kumar [50] proposed Intelligent Data Distribution (*IDD*) and Hybrid Distribution (*HD*), which build upon *CD* and *DD*. *IDD* makes the observation that requiring every node to broadcast its data set to every other member in the group is unnecessarily chatty. They propose a ring communication pattern and illustrate its performance benefits. Still, the cost of globally communicating each node's data set does not allow the algorithm to scale to large data sets. The authors note that the global candidate list can be quite large, and thus they propose a second algorithm, *HD*. *HD* is a hybrid algorithm between *CD* and *IDD*. The authors noted that a weakness in *CD* is that the global candidate list may exceed the size of available main memory for a single node. To address this problem, *HD* combines the advantages of both the *CD* and *IDD* algorithms by dynamically grouping processors and partitioning the candidate set accordingly to maintain good load balance. Each group is large enough to hold the candidate list in main memory, and executes the *IDD* algorithm. Between group communications are performed as in *CD*. We evaluate *HD* for large data sets in Chapter 3.

Schuster, Wolff and Trock [93] parallelize Toivonen's sampling algorithm [99] for a distributed setting, combining it with a common partitioning scheme. This scheme [91] makes the observation that all globally frequent itemset is locally frequent in at least one partition. The algorithm shows excellent scale-up. In addition, the experimentation is one of the few existing works to mine data larger than a few gigabytes (they mine 16 GB). However, as discussed prior, there is no bound on the

number of full data set scans, which can be large when the support is low. Also, the size of the sample at low support is large, which reverts the problem back to mining on out of core data structures with (potentially) poor locality.

Cong *et al.* [32] have proposed a sampling based framework for parallel itemset and sequence mining. They first selectively sample the transactions by discarding a fraction of the most frequent items from each transaction. Based on the mining time on the reduced dataset, rest of the computation is divided into tasks. In practice, they found that sample mining times were quite representative of actual mining times when mining equivalence classes (*frequent-one items*). Using these timing results, tasks could be assigned to machines statically, while affording reasonable load balancing. A key drawback of their approach is that they assume all the data is present on every node, which might not be true in many real world scenarios.

Javid and Khokhar [58] parallelize FPGrowth for distributed machines. They use a comparable tree pruning approach. Although the communication strategy appears to be polling, they do reduce local tree sizes by transferring only data required at each node. They do not present details on the communication mechanism and costs. Results are presented for a small data set (about 1 MB) and for up to eight processors on a shared-memory machine. Their load balancing techniques are not discussed. Zaane, El-Hajj and Lu [122] developed a shared-memory parallelization of FPGrowth. They augment the *FPTree* structure to afford parallel counting. The results show scalability to 64 processors (52-fold) when not factoring in disk I/O. The research shows execution times for a 3GB data set.

Zaki, Parthasarathy, Ogihara and Li parallelized Eclat for distributed memory systems [119]. They place TID lists such that communication is minimized. While

effective in projecting the data set, the method tests for all candidates, even those which do not appear in the data set.

Otey and Parthasarathy have explored distributed itemset mining for dynamic data sets [79]. By using a backtracking strategy called *ZigZag*, they compute the set of maximal global frequent itemsets. By comparing the globally frequent itemsets with locally frequent ones, they find itemsets of high contrast. The largest data set explored in this work is 645MB.

It is clear to us that with the recent proliferation of CMPs, most clusters will be clusters of shared-memory multiprocessors. Initial efforts exist [96, 59] which explore data mining on clusters of SMP machines. Jin and Agrawal [59] developed a parallel algorithm for association rule mining on a cluster of SMP machines. They effectively partitioned the workload to make use of multiple CPUs on a node. The work uses the *Apriori* algorithm, and scans the data set proportionally to the largest frequent item. The work also studies large data sets, 8GB on 8 machines.

Sucahyo *et al.* [96], mine dense data sets. The scalability of their approach appears limited to 12 machines, based on their results. The data sets tested are dense based on the percentage of labels in each transaction. The study shows that bit vectors for dense data sets can be an effective representation. It should be noted that the total number of labels is low. For example, the *chess* data set used has only 75 labels.

2.5 Systems Support for Data Mining

Several researchers have looked at the problem of systems support for distributed data mining algorithms. JAM[88], BODHI[64] and Info-sleuth[75] are high level agent-based distributed systems (in user-space) that support such algorithms.

Bailey, Grossman, Sivakumar, and Turinsky developed a data mining framework called Papyrus[47]. Papyrus is a layered system of services for distributed data mining and data intensive scientific applications. They define three key data movements, namely move data (MD), move models (MM), and move results (MR). These three operations facilitate interaction amongst nodes in a cluster. The paper presents results for *C4.5*, a common decision tree classifier. The system is designed to allow large data sets to be approximated with a model, so that data transfer is less expensive (the model being much smaller than the data). The tradeoff is a loss of information.

The Kensington[23], FreeRIDE[60], and Intelliminer[84] systems look at the problem in terms of clients, and compute and storage servers and implemented basic system level services for data transfer and scheduling of parallel tasks.

Parthasarathy and Dwarkadas[83] developed *InterAct*, a framework for sharing and replicating data between a client and a server. This framework supports data placement, data replication, multiple coherency policies (including automatically updating multiple copies). The key idea is that data is dynamic, here called *Active*, and incurs updates at regular intervals. However, it does not contain a service to preplace transactions. It provides serialization, but does not reduce communication costs.

2.6 Leveraging Architectures for Data Mining

Although the research we describe in this chapter is vast, very little has been done to explicitly leverage architectural advancements in hardware. The majority of data mining researchers assume that the principles of locality will suffice to provide serviceable utilization of the underlying hardware. However, many researchers have studied memory performance for other workloads.

Barroso, Gharachorloo and Bugnion [7] study memory performance of three key workloads; online transaction processing (OLTP), decision support systems (DSS), and Web index search. They draw the conclusion that systems optimized for OLTP will not be well-suited for web index search, because any off-chip cache size will not accommodate both effectively. Ding and Kennedy [35] improve cache performance in dynamic applications by reorganizing data placement at runtime. Parthasarathy, *et al.* developed schemes for parallelizing and improving the performance of *Apriori*-style algorithms on SMP systems [87, 86]. This work is the first to illustrate the benefits of improving memory performance in data mining algorithms in both the sequential as well as the parallel setting. Kim, Qin and Hsu characterize the memory performance for several data mining workloads [66]. Zhou and J. Cieslewicz and K. Ross and M. Shah [123] improve the performance of database applications when executing on simultaneous multithreading processors.

Parthasarathy, Zaki and Li [87], studied memory allocation in parallel itemset mining. They show that pointer-based structures in recursive algorithms generate several performance degrading circumstances, such as poor data locality and false sharing. They present allocation mechanisms, and show that even simple changes can improve execution times by a factor of two.

In a previous effort [40], we examined frequent itemset mining on SMT architectures. We show that co-scheduling threads can improve performance when memory latency is an issue.

2.7 Data Mining on Chip Multiprocessors

There is no existing literature addressing the viability of the Cell processor to perform data mining tasks. There has been some recent efforts to investigate the ability of the Cell to compute scientific workloads. Williams *et al.* [107] showed that the Cell is quite capable of executing Fast Fourier Transform, GEMM, and Stencil computations. They demonstrate an execution time improvement over the Itanium processor by a factor of 12.7, 2.7 and 6.1, respectively for these workloads. In addition, they propose modest changes to the architecture of the Cell which would increase these scalability improvements 3-fold. The work does not address data mining workloads.

Kunzman, *et al.* [69] adapted their parallel runtime framework *CHARM++*, a parallel object-oriented C++ library, to provide portability between the Cell and other platforms. In particular, they proposed *the Offload API*, a general-purpose API to prefetch data, encapsulate data, and peek into the work queue.

2.8 Minwise Hashing

Several works use minwise hashing to process data efficiently. Minwise hashing was first proposed by Cohen [30] to estimate the size of transitive closure and reachability sets. Transitive closure of a graph adds an edge from vertex u to vertex v if and only if there exists a path in the graph from u to v. The technique proposed reduced the execution time to discover closures by grouping similar vertices into classes.

Broder and Charikar developed [16] min-wise independent hashing. The work was motivated during their efforts developing the *Alta Vista* search portal. They needed a mechanism to detect duplicates in web pages. The solution was a two-step process. First, a set was generated from each web page by using n-grams. Second, the sets

were sampled using multiple consistent random permutations. These samples were then used to find duplicates. If two pages had highly similar sets, then the probability that they are the same page is high. The paper furthers the idea by providing several proofs. One such proof shows that the intersection of two pages' sets approximates the Jaccard coefficient between them.

The algorithm afforded by Cohen *et al* [31] leverages k-way hashing to discover association rules with high confidence. The authors seek to find interesting implications ($A \rightarrow B$) with very low support, where A and B are both singletons. This restriction allows for column-wise comparisons for rule detection, but only discovers itemsets of length two. In addition to these smaller patterns, we seek to find very long patterns in the data set. Interestingly, because they are seeking to find two similar items, their matrix is hashed orthogonally to ours. Finally, the authors note that increasing their algorithm to more than 3 columns (and hence patterns of length 4 or more) would suffer exponential overheads. However, their techniques clearly exhibit the general scalability of minimum k-way hashing.

Indyk and Motwani [44] . Goinis, Indyk and Motwani [56] used hashing to find nearest neighbors in high dimensional data. They use hashes as signatures to localize data for answering queries in databases. They illustrate the benefits of the technique over sphere/rectangle trees when the number of dimensions is high.

Gibson, Kumar and Tomkins [42] use min hashing to generate large communities in web graphs. They implement a $H(s, c)$ shingling mechanism which effectively uses smaller shingles in the hash matrix to construct high-level signatures. The high-level signatures are then sorted to find similar groups. These groups then represent highly

connected, large web communities. Empirically, they show the technique scales to very large data sets. The work does not find itemsets.

2.8.1 Web Graph Compression

Several existing works are targeted at web graph compression. Randall *et. al* [90] developed a coding scheme for their LINK database. They leverage the similarity of adjacent URLs and the distribution of the difference (gap) between outlinks for a given node.

Boldi and Vigna [11, 10] developed ζ codes, a family of simple flat codes that are targeted at gaps. ζ codes are well suited for compressing power-law distributed data with small exponents (usually in [1.1-1.3]). They achieve high edge compression and scale well to large graphs. They also demonstrate that using ζ codes one can get close to the compression achieved using Huffman codes without using a coding table. They have also published several useful benchmark data sets that we use in this dissertation. On these data sets, in comparison with Huffman codes, ζ codes are shown to have no more than 5% loss in the number of bits used per edge. The typical loss is in the range of 2-3%. Both works [10, 11] point out the need for the graph to reside in RAM. They also require the URLs to be sorted, and are sensitive to the labels assigned to the graph vertices. As a result, they do not intrinsically support incremental graph updates. Since compressing with references has a small window of only a few nodes, these compression schemes do not incorporate community discovery into the compression of the web graph to support native community queries with a reduced memory footprint.

No existing work attempts to compress large graphs using frequent itemsets.

2.8.2 Community Discovery

There has been much work recently on community discovery in the web graph, and its usefulness. Flake et al. [38] define a community on the web as a set of sites that have more links (in either direction) to members of the community than to non-members. They presented an efficient algorithm using a max-flow / min-cut approach. The web has also been shown to self-organize such that highly related web communities can be efficiently identified based solely on connectivity [39]. This observation has been critical in contributing to the success of link-only (or content agnostic) algorithms for finding communities.

Gibson, Kleinberg and Raghavan [41] presented and provided evidence for structure in underlying web communities. They leverage the HITS [67] algorithm to discover communities influencing web topics. HITS generates authorities and hubs in the web based on the posed query. Each page has both scores, with true hubs having high hub scores and low authority scores (and vice versa). An interesting observation made in the work is that larger communities tend to have greater order in their link structure.

Kumar *et. al* [68] present an alternate mechanism to discover bipartite graphs. They discover small graphs, and then grow them using *HITS*[67]. The authors point out that bipartite graph patterns capture communities which may not be found when enumerating cliques. It is common for two competing companies to not link to each other, but third parties will often link to both.

Gibson, Kumar and Tomkins [42] introduced the strategy of sorting multiple fingerprints generated from minwise hashing to cluster large graphs for community discovery. Using this technique they detect large link spam farms, among other web

graph communities, and illustrate the technique's effectiveness and scalability. Most dense subgraph discovery methods [36] are very stringent in their requirements of matches. However for compression a pattern of length two with frequency three is a net gain. This observation presents the need for scalable approximate data mining, as presented in this work. Also, no existing work has attempted to conjoin graph compression with the community discovery process to improve community query response times.

CHAPTER 3

GLOBAL FREQUENT ITEMSET MINING TERASCALE DATA SETS

3.1 Introduction

"Data mining, also popularly referred to as knowledge discovery from data (KDD), is the automated or convenient extraction of patterns representing knowledge implicitly stored or catchable in *large* data sets, data warehouses, the Web, (and) other *massive information repositories or data streams*" [51]. This statement, from a popular textbook, and its variants resound in numerous articles published over the last decade in the field of data mining and knowledge discovery. As a field, one of the emphasis points has been on the development of mining techniques that can scale to truly large data sets (ranging from hundreds of gigabytes to terabytes in size). Applications requiring such techniques abound, ranging from analyzing large transactional data to mining genomic and proteomic data, from analyzing the astronomical data produced by the Sloan Sky Survey to analyzing the data produced from massive simulation runs.

However, mining truly large data is extremely challenging. As implicit evidence, consider the fact that while a large percent of the research describing data mining

techniques often include a statement similar to the one quoted above, a small fraction actually mine large data sets. In many cases, the data set fits in main memory. There are several reasons why mining large data is particularly daunting. First, many data mining algorithms are computationally complex and often scale non-linearly with the size of the data set (even if the data can fit in memory). Second, when the data sets are large, the algorithms become I/O bound. Third, the data sets and the meta data structures employed may exceed disk capacity of a single machine.

A natural cost-effective solution to this problem that several researchers have taken [2, 32, 50, 83, 86, 118] is to parallelize such algorithms on commodity clusters to take advantage of distributed computing power, aggregate memory and disk space, and parallel I/O capabilities. However, to date none of these approaches have been shown to scale to terabyte-sized data sets. Moreover, as has been recently shown, many of these algorithms are greatly under-utilizing modern hardware[40]. Finally, many of these algorithms rely on multiple database scans and oftentimes transfer subsets of the data around repeatedly.

In this chapter, we focus on the relatively mature problem domain of frequent itemset mining[1]. Frequent itemset mining is the process of enumerating all subsets of items in a transactional database which occur in at least a minimum number of transactions. For retail data, it is the precursor phase to association rule mining, first defined by Agrawal and Srikant[3]. Our solution embraces one of the fastest known sequential algorithms (FPGrowth), and extends it to work in a parallel setting, utilizing all available resources efficiently. In particular, our solution relies on an algorithmic design strategy that is cache-, memory-, disk- and network- conscious, embodying the comprehensive notion of *architecture-conscious data mining*[20, 40].

A highlight of our parallel solution is that we only scan the data set twice, whereas all existing parallel implementations based on the well-known *Apriori* algorithm [4] require a number of data set scans proportional to the cardinality of the largest frequent itemset. The sequential algorithm targeted in this work makes use of tree structures to efficiently search and prune the space of itemsets. Trees are generally not readily efficient, when local trees have to be exchanged among processors, because of the pointer-based nature of tree structure implementations. We devise a mechanism for efficient serialization and merging of local trees called *strip-marshaling and merging*, to allow each node to make global decisions as to which itemsets are frequent. Our approach minimizes synchronization between nodes, while also reducing the data set size required at each node. Finally, our strategy is able to address situations where the meta data does not fit in core, a key limitation of existing parallel algorithms based on the pattern growth paradigm[32].

Through empirical evaluation, we illustrate the effectiveness of the proposed optimizations on large synthetic data sets designed to model retail transaction data. Strip marshaling of the local trees into succinct encoding affords a significant reduction in communication costs, when compared to passing the data set. In addition, local tree pruning reduces the data structure at each node by a factor of up to 14-fold on 48 nodes. Finally, our overall execution times are an order of magnitude faster than existing solutions, such as *CD, DD, IDD* and *HD*.

3.2 Challenges

In this section, we first present challenges associated with itemset mining of large data sets in a parallel setting. The frequent pattern mining problem was first formulated by Agrawal *et al.* [1] for association rule mining. Recalling the problem statement from Chapter 2, the goal is to discover all subsets which occur in at least a minimum number of transactions.

For the distributed case, the data set $D{=}D_1 \cup D_2 \cup ... \cup D_n$, $D_i \cap D_j{=}\emptyset$, $i \neq j$, is distributed or partitioned over n machines. Each partition D_i is a set of transactions. An itemset that is globally frequent *must be* locally frequent in at least one D_i. Also, an itemset not frequent in any D_i cannot be globally frequent, and an itemset locally frequent in all D_i must be globally frequent. The goal *of the parallel itemset mining problem* is to mine for set of all frequent itemsets in a data set that is distributed over different machines. Several fundamental challenges must be addressed when mining itemsets on such platforms.

3.2.1 Computational Complexity

The challenge has exponential computational complexity. The complexity primarily arises from the exponential (over the set of items) size of the itemset lattice. In a parallel setting, each machine operates on a small local partition to reduce the time spent in computation. However, since the data set is distributed, global decision making (e.g., computing the support of an itemset) becomes a difficult task. From a practical standpoint, each machine must spend time communicating the partially mined information with other machines. One needs to devise algorithms that will

achieve load balance among machines while minimizing inter-machine communication overheads.

3.2.2 Communication Costs

The time spent in communication can vary significantly depending on the type of information communicated; data, counts, itemsets, or meta-structures. Also, the degree of synchronization required can vary greatly. For example, one of the optimizations to speed up frequent itemset mining algorithms is to eliminate candidates which are infrequent.The method by which candidates are eliminated is called *search space pruning*. Breadth-first algorithms prune infrequent itemsets at each level. Depth-first algorithms eliminate candidates when the data structure is projected. Each projection has an associated context, which is the parent itemset.

When the data is distributed across several machines, candidate elimination and support counting operations require inter-machine communication since a global decision must be reached to determine result sets. There are two fundamental approaches for reaching a global decision. First, one can *communicate all the needed data, and then compute asynchronously*; An alternate strategy is to *communicate knowledge after a number of steps of computation*. In the former strategy, each machine sends its local portion of the input data set to all the other machines. While this approach minimizes the number of inter-machine communications, it can suffer from high communication overhead when the number of machines and the size of the input data set are large. The communication cost increases with the number of machines due to the broadcast of the large data set. In addition to communication overhead, this approach scales poorly in terms of disk storage space. Each machine requires sufficient

disk space to store the aggregated input data set. For extremely large data sets, full data redundancy may not be feasible. Even when it is feasible, the execution cost of exchanging the entire data set may be too high to make it a practical approach.

In the latter strategy, each machine computes local data and a merge operation is performed to obtain global support counts. Such communication is carried out after every level of the mining process. The advantage of this approach is that it scales well with increasing processors, since the global merge operation can be carried out in $O(N)$, where N is the size of the global count array. However, it has the potential to incur a high communication overhead because of the number of communication operations and large message sizes. As the candidate set size increases, this approach might become prohibitively expensive.

Clearly there are trade-offs between these two different approaches. Algorithm designers must compromise between approaches that copy local data globally, and approaches which retrieve information from remote machines as needed. Hybrid approaches may be developed in order to exploit the benefits from both strategies.

3.2.3 I/O Costs

When mining very large data sets, care should also be taken to reduce I/O overheads. Many data structures used in itemset mining algorithms have sizes proportional to the size of the data set. With very large data sets, these structures can potentially exceed the available main memory, resulting in page faults. As a result, the performance of the algorithm may degrade significantly. When large, out-of-core data structures need to be maintained, the degree of this degradation is in part a

function of the temporal and spatial locality of the algorithm. The size of meta-structures can be reduced by maintaining less information. Therefore, algorithms should be redesigned to keep the meta-structures from exceeding the main memory by reorganizing the computation or by redesigning the data structures [40].

3.2.4 Load Imbalance

Another issue in parallel data mining is to achieve good computational load balance among the machines in the system. Even if the input data set is evenly distributed among machines or replicated on each machine, the computational load of generating candidates may not be evenly distributed. As the data mining process progresses, the number of candidates handled by a machine may be (significantly) different from that of other machines. One approach to achieve load balanced distribution of computation is to look at the distribution of frequent itemsets in a sub-sampled version of the input data set. A disadvantage of this approach is that it introduces overhead because of the sub-sampling of the input data set and mining of the sub-sampled data set. Another approach is to dynamically redistribute computations for candidate generation and support counting among machines when computation load imbalance exceeds a threshold. This method is likely to achieve better load balance, but it introduces overhead associated with redistributing the required data.

Parallel itemset mining offers trade-offs among computation, communication, synchronization, memory usage, and also the use of domain-specific information. In the next section, we describe our parallelization approach and its associated optimizations.

No.	Transaction	Sorted Transaction with Frequent Items
1	f, a, c, d, g, i, m, p	a, c, f, m, p
2	a, b, c, f, l, m, o	a, c, f, b, m
3	b, f, h, j, o	f, b
4	b, c, k, s, p	c, b, p
5	a, f, c, e, l, p, m, n	a, c, f, m, p
6	a, k	a

Table 3.1: A transaction data set with $minsup = 3$

3.3 FPGrowth in Serial

FPGrowth proposed by Han *et al* [52] is a state-of-the-art algorithm frequent itemset miner. It draws inspiration from Eclat [117] in various aspects. Both algorithms build meta-structures in two database scans. They both traverse of the search space in depth-first manner, and employ a pattern-growth approach. We briefly describe *FPGrowth*, since it is the algorithm upon which we base our parallelization.

The algorithm summarizes the data set in the form of a prefix tree, called an *FPTree*. Each node of the tree stores an item label and a count, where the count represents the number of transactions which contain all the items in the path from the root node to the current node. By ordering items in a transaction based on their frequency in the data set, a high degree of overlap is established, reducing the size of the tree. *FPGrowth* can prune the search space very efficiently and can handle the problem of skewness by projecting the data set during the mining process.

A prefix tree is constructed as follows. The algorithm first scans the data set to produce a list of frequent 1-items. These items are then sorted in frequency descending order. Following this step, the transactions are sorted based on the order from the second step. Then, infrequent 1-items are pruned away. Finally, for each transaction, the algorithm inserts each of its items into a tree, in sequential order, generating new

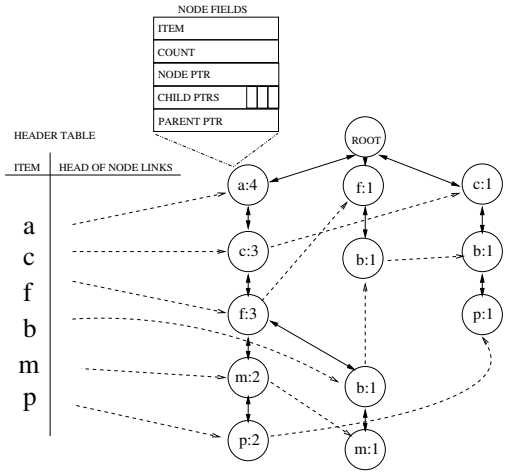


Figure 3.1: An FP-tree/prefix tree

nodes when a node with the appropriate label is not found, and incrementing the count of existing nodes otherwise.

Table 3.1 shows a sample data set, and Figure 3.1 shows the corresponding tree. Each node in the tree consists of an *item*, a *count*, a *nodelink ptr* (which points to the next item in the tree with the same item-id), and *child ptrs* (a list of pointers to all its children). Pointers to the first occurrence of each item in the tree are stored in a header table.

The frequency count for an itemset, say ca, is computed as follows. First, each occurrence of item c in the tree is determined using the node link pointers. Next, for each occurrence of c, the tree is traversed in a bottom up fashion in search of an occurrence of a. The count for itemset ca is then the sum of counts for each node c in the tree that has a as an ancestor.

3.4 Parallel Optimizations

In this work, we target a distributed-memory parallel architecture, where each node has one or more local disks and data exchange among the nodes is done through message passing. The input data set is evenly partitioned across the nodes in the system. We first present our optimizations, and then in Section 3.4.4 we discuss how these optimizations are combined in the parallel implementation.

3.4.1 Minimizing Communication Costs

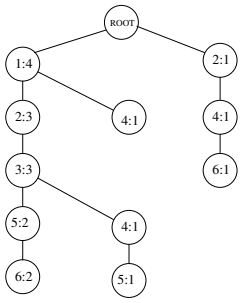
One of the fundamental challenges in parallel frequent itemset mining is that global knowledge is required to prove an itemset is not frequent. It can be shown that if an itemset is locally frequent or infrequent in every partition, then it is globally frequent or infrequent, respectively. However, in practice, most itemsets lie in between; they are locally frequent in a nonempty proper subset of the partitions and infrequent in the remaining partitions. These itemsets must have their exact support counts from each partition summed for a decision to be made. In Apriori-style algorithms, this is not an additional constraint, because this information is present. Apriori maintains exact counts for each potentially frequent candidate at each level. However, *FPGrowth* discards infrequent itemsets when projecting the data set.

It is too expensive for a node to maintain the count for every unsupported itemset so that it can be retrieved by another node. This would equate to mining the data set at a support of one. Alternatively, synchronizing at every pass of every projection of each equivalence class is also excessive. Our solution is for each machine to communicate its local *FPTree* to all other machines using a ring broadcast protocol. That is, each node receives an *FPTree* from its left neighbor in a ring organization,

merges the received tree with its local *FPTree*, and passes the tree received from its left neighbor to its right neighbor.

Exchanging the *FPTree* structure reduces communication overhead because the tree is typically much smaller than the frequent data set (due to transaction overlap). To further reduce the volume of communication, instead of communicating the original tree, each processor transfers a concise encoding, a process we term *strip marshaling*. The tree is represented as a array of integers. To compute the array (or encoding), the tree is traversed in depth first order, and the label of each tree node is appended to the array in the order it is visited. Then, for each step back up the tree, the negative of the support of that node is also added. We use the negative of the support so that when decoding the array, it can be determined which integers are tree node labels and which integers are support values. Naturally, we again negate the support value upon decoding, so that the original support value is used.

This encoding has two desired effects. First, the encoded tree is significantly smaller than the entire tree, since each tree node can be represented by two integers only. Typically, *FPTree* nodes are 40 bytes (two 4-byte ints and four 8-byte pointers, as shown in Figure 3.1, where sibling pointers are linked lists). The encoding requires only 8 bytes, resulting in a 5-fold reduction in size. In many cases, the label can be represented as a short, resulting in a 6.3-fold reduction. However, this will require words which are not aligned on word boundaries, which incurs additional runtime costs. Second and more importantly, because the encoded representation of the tree is in depth-first order, each node can traverse its local tree and add necessary nodes online. Thus, merging the encoded tree into the existing local tree is efficient, as it only requires one traversal of the each tree. When the tree node exists in the



ENCODING (RELABELED):
1,2,3,5,6,–2,–2,4,5,–1,–1,–3,–3,4,–1,–4,2,4,6,–1,–1,–1

Figure 3.2: *Strip Marshaling* the *FPTree* provides a succinct representation for communication.

encoding but not in the local tree, we add it to the local tree (with the associated count); otherwise we add the received count (or support) to the existing tree node in the local tree. Because the processing of the encoded tree is efficient, we can effectively overlap communication with processing. Also, transferring these succinct structures dispatches our synchronization requirements. The global knowledge afforded allows each machine to subsequently process all its assigned tasks independently.

For example, Figure 3.2 illustrates the tree encoding for the tree from our example in Section 3.2. Note that the labels of the tree nodes are recoded as integers after the first scan of the input data set.

3.4.2 Pruning Redundant Data

Although the prefix tree representation (*FPTree*) for the projected data set is typically quite concise, there is not a guaranteed compression ratio with this data structure. In rare cases, the size of the tree can meet or exceed the size of the input data set. A large data set may not fit on the disk of a single machine. To alleviate

this concern, we prune the local prefix tree of unnecessary data. Specifically, each node maintains the portion of the *FPTree* required to mine the itemsets assigned to that node.

Recall that to mine an item i in the *FPTree*, the algorithm traverses every path from any occurrence of i in the tree up to the root of the tree. These upward traversals form the projected data set for that item, which is used to construct a new *FPTree*. Thus, to correctly mine i only the nodes from occurrences of i to the root are required.

Let the items assigned to machine M be I. Consider any $i \in I$, and let n_i be a node in the *FPTree* with its label. Suppose a particular path P in the tree of length k is

$$P \quad = \quad (n_1, n_2, ..., n_i, n_i + 1, ..., n_k). \tag{3.1}$$

To correctly mine item i, only the portion of P from n_1 to n_i is required for correctness. Therefore, for every path P in the *FPTree*, M may delete all nodes n_j occurring after n_i. This is because the deleted items will never be used in any projection made by M. In practice, M can start at every leaf in the tree, and delete all nodes from n_k to the first occurrence of any item $i \in I$ assigned to M. M simply evaluates this condition on the *strip marshaled* tree as it arrives, removing unnecessary nodes.

As an example, we return to the *FPTree* described in Section 3.2. Assume we adopt a round-robin assignment function, so machine M is assigned all items i such that

$$i \quad \% \quad |Cluster| \quad = \quad rank(M). \tag{3.2}$$

We illustrate the local tree for each of the four machines in Figure 3.3. A nice property of this scheme is that as we increase the number of machines in the cluster, we increase the amount of pruning available.

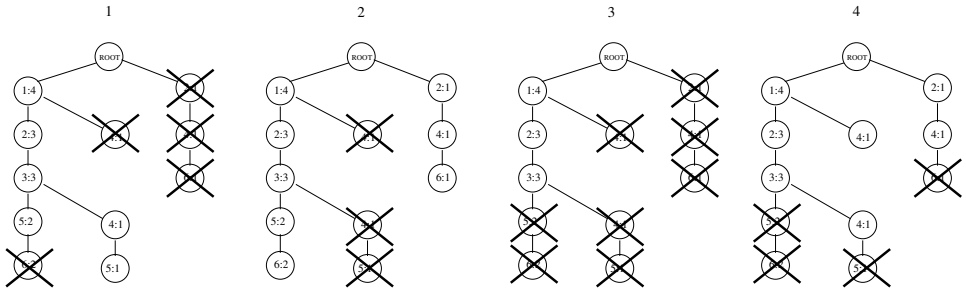


Figure 3.3: Each node stores a partially overlapping subtree of the global tree.

3.4.3 Partitioning the Mining Process

We employ a static allocation strategy to assign data mining tasks to the nodes in the system, because it affords data reduction, as shown in our tree pruning procedures. We determine the mapping of tasks to machines before any mining occurs. Researchers[32] have shown that intelligent allocation using sampling often leads to improved load balancing with low process idle times, in particular with *FPGrowth*. We leverage such sampling techniques to assign *frequent-one* items to machines after the first parallel scan of the data set. We note that this technique is more effective for data sets with balanced associativity in the the label space because it usually results in less load imbalance in the mining phase.

3.4.4 Putting It All Together

We now use the optimizations above to construct our solution. The approach is shown in Algorithm 1, and we label it DFP (Distributed FPGrowth).

The machines are logically structured in a ring. In phase one (lines 1 - 5), each machine scans its local data set to retrieve the counts for each item. Lines 1 through 3 count the frequency of each label in the data set. In line 4, each machine then sends

its count array to its right-neighbor in the ring, and receives the counts from its left neighbor. This communication continues *n-1* times until each count array has been witnessed by each machine. In line 5, the machines then perform a voting procedure to statically assign frequent-one items to particular nodes. The mechanism for voting is to mine a very small sample of the data set (about 1%) to estimate the cost of mining each frequent-1 item. This step is designed to minimize load imbalance.

In phase two (lines 6 - 12), each machine builds a local tree with sufficient global information to mine each assigned frequent-1 item. In line 7, each machine builds a local prefix tree. In line 8, each machine encodes the local tree as an array of integers. In line 10, these arrays are circulated in a ring manner, as was performed with the count array. Since each machine requires only a subset of the total data to mine its assigned elements, upon receiving the array the local machine incorporates only the contextually pertinent parts of the array into its local tree before passing it to its right-neighbor. In line 11, the local data is furthered pruned. This is necessary because the first tree had full local data, which is required to generate the array, but unnecessary for local computation. In phase three (lines 12 - 13), each machine independently mines its assigned items. No synchronization between machines is required during the mining process. The results are then aggregated in line 14.

3.5 Experiments

We implement our optimizations in C++ and MPI. The test cluster has 64 machines connected via Infiniband, 48 of which are available to us. Each machine has two 64-bit AMD Opteron 250 single core processors, 2 RAID-0 250GB SATA hard drives, 8GB of RAM, and runs the Linux Operating System. We are afforded 100GB

Algorithm 1 DFP

Input: Data set D=$D_1 \cup ... \cup D_n$, *Global* Support σ
Output: F = Set of frequent itemsets

```
1: for i = 1 to n do
2:    Scan D_i for item frequencies, LF_i
3: end for
4: Aggregate LF_i, i=1..n (using a ring)
5: Assign itemsets to nodes
6: for i = 1 to n do
7:    Scan the D_i to build a local Prefix Tree, T_i
8:    Encode T_i as an array, S_i
9: end for
10: Distribute S_i, i=1..n (using a ring)
11: Prune T_i of unneeded data
12: Locally mine the pruned global tree
13: Aggregate the final results (using a ring)
```

(per machine) of the available disk space. For this work, we only use one of the two available processors. All synthetic data sets were created with the IBM Quest generator, with 100,000 distinct items, an average pattern length of 8, and 100,000 patterns. Other settings were defaults. The number of transactions varies depending on the experiment, and will be expressed herein. Our main data set is 1.1 terabytes, distributed over 48 machines (24GB each), called 1TB. The real data sets we use, namely *Kosarak, Mushroom and Chess*, are available in the FIMI repository [43]. Their average transaction lengths are 8, 23 and 37, respectively.

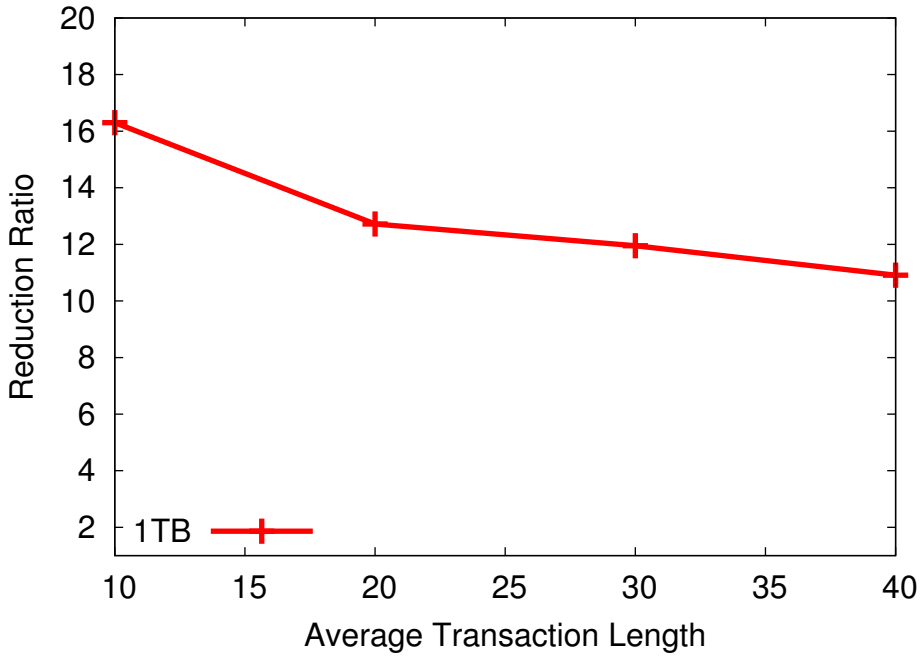
As a comparison, we have implemented *CD, DD*[2] and *IDD*[50]. We should point out that CD never exceeded the available main memory of a machine, for all data sets we evaluated (due to the large amount of available main memory per machine). As a result, *HD*[50] is never better than CD.

3.5.1 Evaluating Storage Reduction

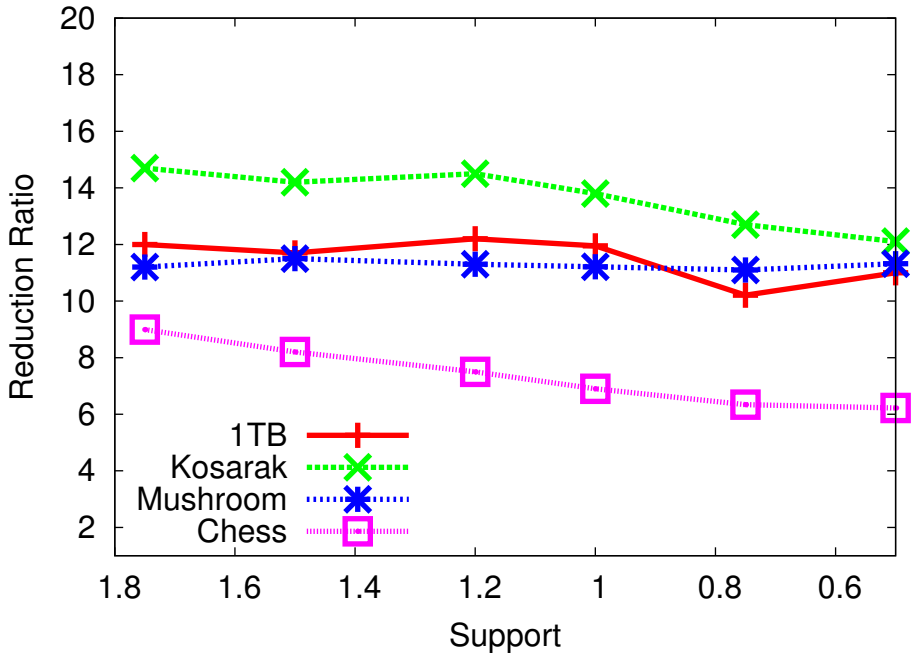
To evaluate the effectiveness of pruning the prefix trees, we record the average local tree size and compare against an algorithm that maintains the entire tree on each machine. We consider the performance of pruning with respect to three parameters, namely the transaction length, the chosen support and the number of machines (see Figures 3.4 and 3.5).

First, we consider transaction length. Since pruning trims nodes between the leaves and the first assigned node, we suspect that increasing transaction length will degrade the reduction. The number of machines for these experiments was 48, and the support was set to be 0.75%. The data set was created with the same parameters as 1TB, except that the transaction length is varied. It can be seen that performance degrades gradually from 16-fold to 12-fold when increasing transaction lengths from 10 to 40 on synthetic data. Degradation in pruning was dampened by the average frequent pattern length, which was not altered. As shown in Figures 3.4 (bottom) and 3.5, real data sets with longer average transaction lengths generally exhibit less pruning. Recall, the three real data sets have average transaction lengths of 8 (Kosarak), 23 (Mushroom) and 37 (Chess).

We next consider the impact of support on tree pruning. In Figure 3.4 (bottom), we vary the support from 1.8% to 0.5% for Kosarak and 1TB, from 18% to 5% for Mushroom, and from 40% to 5% for Chess. The difference in chosen supports is due to the differences in data set densities. In all cases, we use 48 machines. It can be seen in the figure that support only slightly degrades the reduction ratio. This is because although some paths in the tree increase in length (and are thus less prunable), more

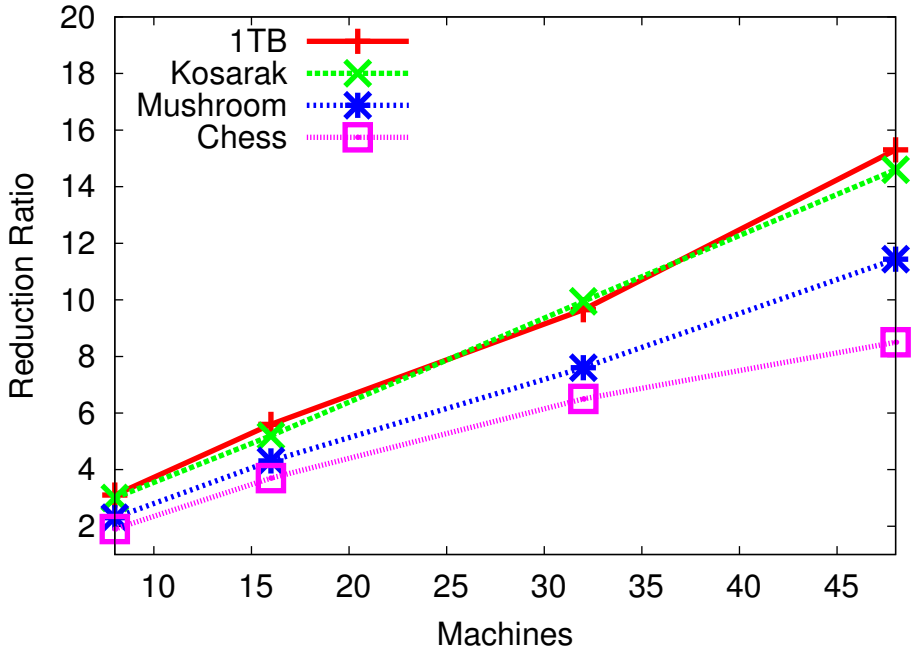


(a) Reduction vs. transaction length



(b) Reduction vs. minimum support

Figure 3.4: Reduction afforded by tree pruning as a function of transaction length and minimum support.



(a) Reduction vs. cluster size

Figure 3.5: Reduction afforded by tree pruning as a function of cluster size.

short paths are introduced, which afford increased prunability. Therefore, as we lower support, the technique continues to prune effectively.

Finally, we evaluate pruning with respect to number of machines used. From Figure 3.4, it can be seen that the relative amount of reduction increases with an increasing number of machines. For example, at 8 machines we have a 3-fold reduction, but using 48 machines it increases to a 15-fold reduction. We see more improvement from this optimization as the number of machines is increased because each machine has fewer allocated items, thus more tree nodes may be removed from the paths. We found that the variance in local tree size is larger for real data than synthetic data.

3.5.2 Evaluating Communication Reduction

To evaluate the cost of communication, we perform a weak scalability study. Using 8 machines, we mine 1/6th of the data set, and using 48 machines we mine the entire 1TB data set. Each machine contains a 24GB partition. We are interested in measuring the amount of network traffic generated by communicating local trees. Table 3.2 displays our results, mining at two different supports. The column labeled *Glbl T'* is the size of the local tree if we maintain full global knowledge at each node. The column labeled *Lcl T* is the average size of the local tree if we maintain only local data at each node. The column labeled *Lcl T'* is the average size of the local tree if we maintain partial global knowledge at each node.

Several issues are worth noting. First, the aggregate size of communication for the process (total transferred) is much smaller than the size of the data set. For example, at a support of 0.75% on 48 machines, we transfer about 19 gigabytes instead of the full 1152GB. *This results in more than a 60-fold reduction in data transferred*

Machines	Data(GB)	supp%	Glbl T'(GB)	Lcl T'(GB)	Lcl T(GB)	encdng(MB)	transferred(GB)
8	192	1.00	1.66	0.54	0.32	91.63	0.72
16	384	1.00	3.23	0.57	0.32	96.75	1.51
32	768	1.00	4.87	0.55	0.31	93.33	2.92
48	1152	1.00	6.77	0.58	0.32	98.77	4.63
8	192	0.75	7.33	2.37	1.65	403.63	3.15
16	384	0.75	13.59	2.38	1.63	406.87	6.36
32	768	0.75	21.31	2.39	1.65	408.58	12.77
48	1152	0.75	27.78	2.37	1.65	405.16	18.99

Table 3.2: Communication costs for mining 1.1 terabytes of data (data set 1TB).

when compared to transferring the entire local data set. From an execution time standpoint, the 60-fold reduction reduces the communication phase from a worst-case of 21 minutes to 35 seconds.

Second, the prefix tree affords a succinct representation of the data set. For each transfer, we save 24GB/2.39GB=10-fold due to the tree structure. Third, tree encoding affords an additional communication reduction. We save an additional 6-fold due to this *strip marshaling.* Also, we make the transfer only once, whereas *IDD* must perform the transfer at each level (at this support, that is 11 levels in all). We do note that in some cases, such as the 8-machine case, it may be possible to improve the excessive traffic of *IDD* by writing some of the data to local storage. At the very least, these methods will require more than 60-fold more data transferred. For large data sets, this increase in transfer time is a significant percentage of the total execution time.

3.5.3 Evaluating Weak Scalability

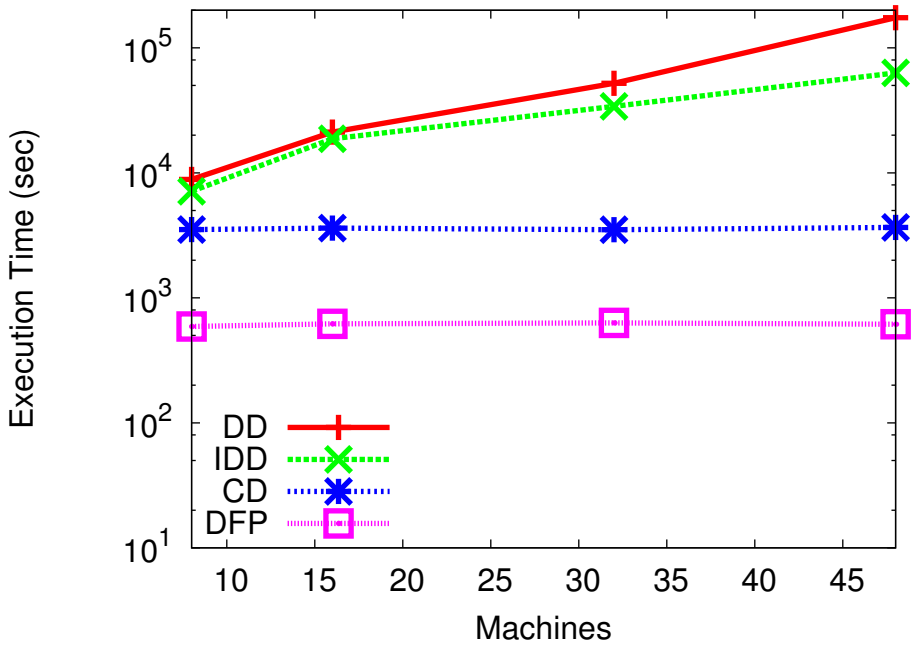
To evaluate weak scalability, we employ the same experimental setup. We consider execution time performance as the data set is progressively increased proportional to the number of machines. We mined this data set at two different support thresholds,

as shown in Figure 3.6. The latter support required significantly larger data structures (see Table 3.2). *DD* suffers greatly from the fact that it transfers the entire data set to each machine at each level. These broadcasts scale poorly with increasing cluster size. For example, at 8 machines there are 8 broadcasts of 24GB. Scaling to 48 machines, *DD* must make 48 independent broadcasts of 24GB, all of which must be sent to 47 other machines. The Infiniband 10Gbps interconnect is not the bottleneck, as each machine is limited by the disk bandwidth of the sender, which is only approximately 90Mbps.

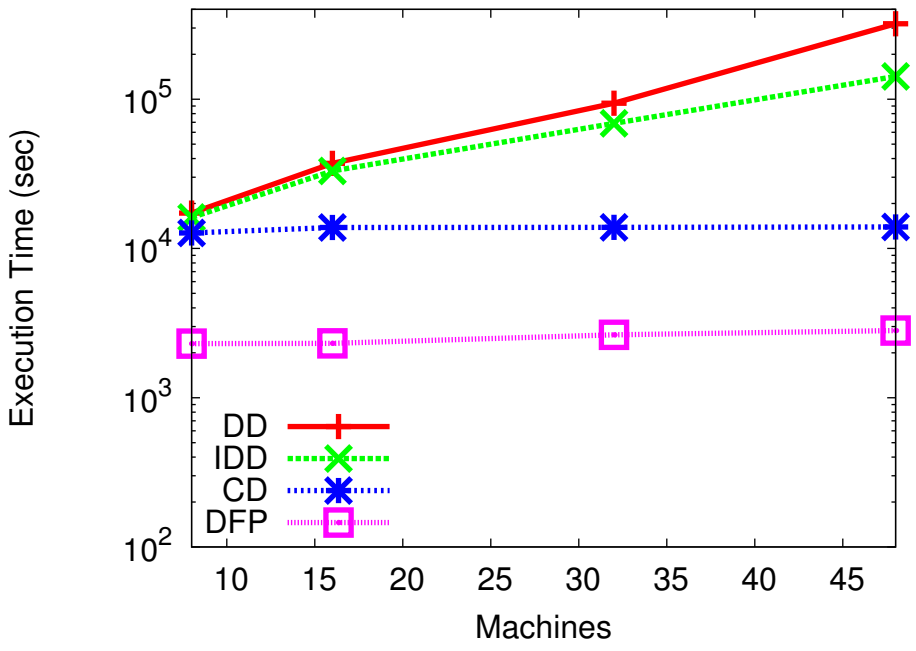
IDD improves upon *DD* slightly, but the inherent challenge persists. *IDD* uses a ring to broadcast its data. However, it makes little effort to minimize the size of the communicated data, and thus scales poorly on large data sets.

CD is a significant improvement over *IDD*. *CD* performs the costly disk scans in parallel. *CD* pays a penalty due to its underlying serial algorithm, which requires an additional scan of the entire data set at each level of the search space. However, *CD* shows excellent scalability, since it parallelizes the largest portion of the execution time. The algorithm proposed in this chapter, *DFP*, performs most efficiently. It leverages the available memory space by building a data structure which eliminates the need to repeatedly scan the disk. Also, it effectively circulates sufficient global information to allow machines to mine tasks independently, but does not suffer the penalties that are incurred by *DD* and *IDD*.

Some small overhead (<5%) is incurred when the number of machines becomes large because disk bandwidth heterogeneity exists, and all algorithms presented scan the data set from disk at least twice.



(a) sup=1.0%



(b) sup=0.75%

Figure 3.6: Weak scalability for 1TB at support = 1.0 and 0.75%.

3.5.4 Evaluating Strong Scalability

To evaluate strong scalability (or speedup), we partitioned 500GB of synthetic data onto 8, 16, 32 and 48 machines. Thus the amount of data in all trials is 500GB. We then ran the algorithms at various supports. Because of the regularity in the data, the descrease in support roughyl constitutes a linear increase in work. The results are available in Figure 3.7. Speedups are in relation to the time to mine the data set on eight machines, since we could not fit 500GB on a smaller number of machines. We do not include *DD* or *IDD* in the figure because their performance was extremely poor, as they do not improve the time to transfer the data between machines. For example, on 8 machines *DD* requires 140 minutes just to pass the data to each node. This number does not improve with an increasing number of machines because although the data on each machine is less, more machines must transfer their data to more locations. Therefore, *DD* and *IDD* do not scale well on large data sets.

CD scales admirably, from 391 to 80 minutes when increasing the cluster size from 8 to 48 machines. *DFP* scales even better, from 39.3 to 7.5 minutes on the same sized clusters. Also, *DFP* is about 10-fold faster. We believe this gap will widen when mining dense data sets, because *CD* does not parallelize the candidate generation phase.

3.6 Conclusion

In this chapter, we have presented a parallelization of a fast frequent itemset mining algorithm to enable efficient mining of very large retail data sets. Our implementation employs an "architecture-conscious" design strategy to efficiently utilize

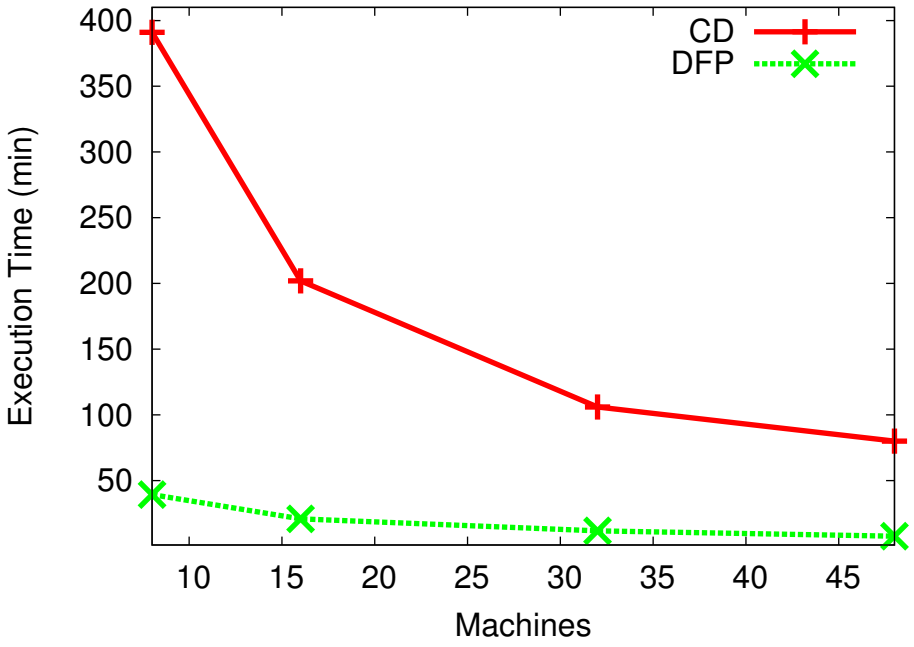


Figure 3.7: Speedup for 500GB at support = 1.0%.

memory, processing, storage, and networking resources while minimizing communication overheads. The salient features include i) a serialization and merging strategy for computing global trees from local trees, ii) efficient tree pruning for better use of available memory space, and iii) the fact that the input data set is scanned only twice. Our experimental evaluation using a state-of-the-art commodity cluster and large data illustrates a speedup of an order of magnitude when compared to an efficient parallel algorithm, *CD*. The pruning strategy is able to reduce the size of the data by up to 16 times on 48 nodes, and the amount of reduction improves as the number of nodes is increased. Also, the serialization strategy results in a six-fold reduction when exchanging local tree data between nodes.

In Chapter 6, we will explore mining itemsets from irregular, power law distributed data such as large web graphs. These workloads propose additional challenges because the data has a greater degree of imbalance, it has higher associativity, and the use cases require the mining process to evaluate far lower supports.

CHAPTER 4

PARALLEL FREQUENT SUBSTRUCTURE MINING ON EMERGING PARALLEL PROCESSORS

4.1 Introduction

Computer architectures are experiencing a revolution in design. Major modifications such as Chip Multiprocessing (CMP), simultaneous multithreading (SMT), and 64-bit computing afford significant advancements in processing power, and also provide true MIMD (multiple instruction multiple data) parallel computing for desktop PCs. SMT provides support for multiple contexts to compete simultaneously for chip resources, in an effort to improve overall utilization. It is essentially a hardware-based latency-hiding technique. CMP designs (also known as multicore architectures) incorporate more than one processing element on the same die. They can execute multiple threads concurrently because each processing element has its own context *and* functional units.

CMPs arise in part because of the inherent challenges with increasing clock frequencies. The increase in processor frequencies over the past several years has required a significant increase in voltage, which has increased power consumption and heat dissipation of the central processing unit. In addition, increased frequencies require

considerable extensions to instruction pipeline depths. Finally, since memory latency has not decreased proportionally to the increase in clock frequency, higher frequencies are often not beneficial due to poor memory performance. By incorporating thread level parallelism, chip vendors can continue to improve IPC by exploiting thread level parallelism, without raising frequencies. It is clear that the number of cores on a chip will rise significantly over the next several years. Intel recently released a two core chipset named the *PentiumD*®– a 2.4GHz version costs under $250. Last year, Sun Microsystems has released the *Niagara*®, an eight core CPU capable of running 32 concurrent threads. [3]. Sony, Toshiba, and IBM (STI) developed a special-purpose 9 core CMP called the Cell Processor for the Sony Playstation 3. It is designed to improve computational throughput for streaming applications, and will be discussed in detail in Chapter 5. For the general-purpose MIMD CMPs, parallelization challenges share many similarities with traditional single-core, multiple-processor machines (Symmetric Multiprocessing machines, or simply SMPs). However, with CMP architectures the cores not only share all off-chip resources, but some on-chip resources, such as cache. This sharing of resources makes efficient use of the afforded CPU cycles quite challenging. Therefore, *it is paramount that algorithm designers research effective strategies to incorporate task parallelism and efficient memory system usage to ensure that application performance is commensurate with such architectural advancements.*

Recently the frequent itemset mining problem statement has been generalized from transaction data to graph data. Finding frequent patterns in graph databases such as chemical and biological data [82], XML documents [98], web link data [89],

[3]Each core on the processor can execute four concurrent threads due to Simultaneous Multithreading

financial transaction data, social network data, and other areas has posed an important challenge to the data mining community. Representing these data sets in graphical form can capture relationships among entities which are lost when the data is represented as transactions. For example, in web session data, frequent patterns represent resources which are more often used, and the paths most often taken to access these resources. Application service providers can improve web server behavior by prefetching resources as users move the web application according to their typical session patterns. In molecular data sets, nodes correspond to atoms, and edges represent chemical bonds. Frequent substructures may provide insight into the behavior of the molecule. For example, graphs containing benzene rings connected to H-C=O groups tend to have a pleasant odor. Frequent substructure mining can be used to group previously unknown groups, affording new information. Frequent substructure mining can be used to group previously unknown groups, affording new information. Zaki et al [121] apply graph-based methods on weighted secondary graph structures of proteins to predict folding sequences. Finally, in our financial transaction data set, large graphs with high occurrence rates could represent potential criminal activity, such as money laundering.

A major challenge in substructure mining is that the search space is exponential with respect to the data set, forcing runtimes to be quite long. A simple chemical data set of 300 molecules can require many hours to mine when the user specifies a low support threshold. One reason for the inherent difficulty is that identifying the embedding of one graph in another (*subgraph isomorphism*) is computationally expensive. As such, leveraging architectural advancements is paramount to solving these problems efficiently. The problem of excessive runtimes is compounded by the

fact that data mining is an interactive and iterative process, requiring the user to query the data set many times to obtain end results. By redesigning graph mining algorithms to leverage the task level parallelism abilities of CMP processors, graph mining runtimes can be significantly reduced.

In this work, we present a parallel graph mining algorithm designed to adapt its runtime behavior based on system-level feedback, so as to use available resources as efficiently as possible. It is based on the serial algorithm *gSpan*. We target shared-memory multiprocessors, namely as CMPs and SMTs, in an effort to solve the challenges outlined above. Most notably, we address the need to effectively balance the load in the system. Specifically, the contributions of this chapter are

- we describe several key issues to consider when developing algorithms for CMP architectures
- we provide a scalable parallel graph mining algorithm for CMP and other shared memory systems
- we present several queuing models and show that a distributed model affords the highest scalability
- we present an adaptive partitioning mechanism to increase computational throughput
- we describe a mechanism to leverage addition memory for shared memory processors, when it is available and its use has utility.

The remainder of the chapter is organized as follows. In Section 4.2 we provide background text on graph mining and the relevant research. We examine the literature

and choose a baseline algorithm for parallelization. In Section 4.3 we examine the challenges with developing an efficient, scalable parallel graph mining algorithm. We also discuss the challenges related to leveraging CMPs effectively. In Section 4.4 we describe our algorithm and optimizations. We evaluate these optimizations in Section 4.5, discuss their implication in Section 4.6, and conclude in Section 4.7.

4.2 Background

In this section, we examine the problem of frequent substructure mining and its existing literature. Through an understanding of the current research, we can begin to develop an efficient parallel algorithm. We first begin with a brief problem statement.

4.2.1 Problem Statement

We generalize the problem statement for frequent itemset mining presented in Chapter 2 for graph data. A graph is a set of vertices V and a set of edges $E \subseteq (V \times V)$, where E is unrestricted (graphs can have cycles). Nodes and edges may have labels, and may be directed or undirected. Let us define the labeling function $l : (E, V) \rightarrow L$, where all elements of E and V are mapped to exactly one label. A graph g' is a subgraph of g if there is a bijection from the edges $E' \subseteq E$ with matching labels for nodes and edges. This mapping is also called an embedding of g' in g. If both g' and g are subgraphs of each other, they are considered isomorphic. A subgraph g' is frequent if it there is an embedding in at least σ graphs in the database, where σ is a user-defined threshold, or $\#(g \mid g \in G \mid g' \subset g) \geq \sigma$. The problem is then to enumerate all frequent graphs g' in D. Note that this problem definition has only one deterministic solution set. There is no 'goodness' measure to a frequent subgraph g'; it either is a subgraph of at least σ database graphs, or it is not.

Algorithm	Time	Memory	Parallelism
FFSM	3	5	3
FSG	5	2	3
GastonEL	1	4	3
gSpan	2	1	1
MoFa	4	5	2

Table 4.1: Serial Algorithm Study.

4.2.2 Selecting a Baseline

Several serial algorithms exist that enumerate frequent subgraphs. FSG [70] mines for all frequent subgraphs using an *Apriori* breadth-first fashion. *gSpan*, developed by Yan and Han [110], affords a significant computational improvement. Another recent serial graph miner is *GastonEL* [78], developed by Nijssen and Kok. *GastonEL* incorporates embedding lists using an efficient pointer-based structure. This provides an improvement in execution time at the cost of significantly increased memory consumption. To select a baseline algorithm, we evaluated five efficient frequent graph miners, presented in Table 4.1.

GastonEL displayed the lowest execution times on average, with *gSpan* second. For data sets with result sets containing a large percentage of trees, *GastonEL* was faster than *gSpan*. This can be attributed to its efficient canonical labeling for trees. When the frequent result set was dominated by cyclic graphs, *GastonEL* and *gSpan* had about the same execution times. However, *GastonEL* consumes a large amount of memory. Further, *GastonEL*'s use of embedding lists hinders parallelization because child tasks share data structures. *gSpan* consumes very little memory because it does not use embedding lists. The other three algorithms we evaluated did not provide

an improvement in either memory consumption or execution time when compared to these two algorithms. An independent study by Wörlein *et al* [109], generally had similar findings. In light of our rankings, we chose *gSpan* as a baseline for our parallel algorithm. We also parallelized *GastonEL* as part of this study, using source code provided by the original authors [78]. The resulting scalability was lower than what we present in Section 4.5 due to the static embedding lists.

Algorithm 2 gSpan

Input: Global Support threshold σ
Input: Graph Data set D
Output: Set of frequent subgraphs S

```
1: Count the occurrences of all edge and node labels in D
2: Count the occurrences of all edge and node labels in D
3: Remove infrequent edges and nodes
4: Sort and relabel remaining nodes and edges
5: for each Frequent-1 edge e ∈ D do
6:     Initialize DFScode s with e
7:     D_e = Graphs in D which contain e
8:     Remove e from all graphs in D
9:     SubMine(D_e,s)
10: end for
```

Algorithm 3 SubMine

Input: Projected Graph Dataset D
Input: DFScode parent
Output: $S = S \cup f(parent)$

```
1: if parent is not the canonical label then
2:    return
3: end if
4: Add parent to the result set
5: EL = All embeddings of parent in D
6: C = All child extensions to parent in EL
7: for each Child c in C do
8:    if support(c) ≥ σ then
9:       DFScode s = parent+c
10:   end if
11:   D_s = Subset of D containing s
12:   SubMine (D_s,s)
13: end for
```

As our optimizations are based on *gSpan*, we now describe the algorithm in detail. Given a data set D, all edges e which occur in at least σ graphs ($g \in D$) are used as seeds to partition the search space. These are called *frequent-1 edges.* Each frequent-1 edge is grown with all candidate edges. A candidate edge is any other frequent edge, which either grows the graph by adding an additional node, or merely creates a new cycle by adding just an edge. This simple process is then recursed upon, until the currently grown graph has no frequent candidates for growing. *gSpan* only grows graphs with edges that result in subgraphs known to be in the data set, thus reducing unnecessary work. *gSpan* represents a graph with a labeling system called DFScodes. A DFScode for a graph is generated by outputting newly discovered nodes during a depth-first traversal of a graph. For each edge taken, a five-tuple is added to the DFScode, namely *(srcNode, destNode,srcLabel, edgeLabel, destLabel).* A graph may have many valid DFScodes.

gSpan restricts growth to the right-most path of the DFS tree. In addition, a cycle-closing refinement (a backedge) can only be grown from the right-most node. While these restrictions greatly reduce the possibility of redundant graph mining, it is not completely avoided. Therefore, *gSpan* verifies that each subsequent DFScode is the canonical code for the graph it represents prior to mining it, thus avoiding duplicate work. This operation is done by performing a depth-first walk from the lexicographically smallest edge, and at each step in the walk, taking the smallest possible edge. If there are multiple choices which are equal, both are checked. If a valid DFScode for the graph can be produced which is smaller (lexicographically) than the DFScode in question, the check fails. To mine each possible child graph, the data set graph IDs of the parent are used to search for child embeddings. *gSpan* rediscovers the entire new child graph for each recursive mining call. For example, if graph *(A-B-C)* was found in data set graphs 1, 2 and 3, and there is sufficient support for child candidate *(-D)*, *gSpan* will rediscover the parent *(A-B-C)* when mining *(A-B-C-D)*. In other words, *gSpan* does not maintain the mappings between recursive calls.

Pseudo code for *gSpan* is provided in Algorithms 2 and 3. In the first algorithm, lines 1-4 remove infrequent edges from the graph, where an edge is a 3-tuple *(srcLabel,edgeLabel,destLabel)*. This requires an additional data set scan but can significantly reduce the size of the data set. Then, in lines 5-7 each frequent-1 item is passed to 3. In 3, the DFScode is evaluated to verify it is the canonical label for the graph it represents. For example, suppose we have graph *A-B-C*. There are four possible DFScodes for this graph, although only the DFS walk which starts with *A* will be the canonical representation. Next, the data set is scanned to find all embeddings of the

graph. Once these embeddings are found, they are traversed to determine the list of possible child extensions. Each extension which occurs at least the minimum number of times is recursed upon. Further details of this algorithm can be found elsewhere [110].

4.3 Challenges

In this section, we examine the challenges associated with efficient parallelization of graph mining on CMPs.

4.3.1 Chip Multiprocessing Challenges

Chip Multiprocessing (CMP) designs (often called multicore) incorporate more than one processing element on the same die. They can execute multiple threads concurrently because each processing element has its own context and functional units. CMP systems have significant differences when compared to existing platforms. To begin with, *inter-process communication costs can be much lower than previous parallel architectures*. Distributed clusters must exchange messages, and traditional shared memory multiple CPU systems write to main memory. CMP systems can keep locks on chip, either in specially designed hardware, or in a shared L2 cache. A direct consequence of lower locking costs is an improved task queuing system. Because CMP chips are designed for thread level parallelism, low-overhead software queues are essential; on chip hardware-based queues are also attractive. Therefore, *algorithm designers should consider a much finer granularity* when targeting these systems. Single-chip CMP systems typically have less memory and cache per processor than high-end parallel architectures such as message-passing clusters or shared memory multiprocessors. As such, algorithms which process large data sets in parallel on

these systems must take care to minimize the use of main memory. In addition to simply exceeding main memory, excessive memory use can also lead to bus contention. This is exacerbated with CMPs, since processing elements share memory bandwidth. We studied memory traffic on an 16-way SMP, and found that when increasing from 1 to 16 processors, bus utilization rose from 0.2% to 25%. Algorithm designers can compensate by improving thread-level data sharing. *We believe that algorithms which trade slightly increased computation for lower memory consumption will exhibit better overall performance on CMP systems*, when compared to systems which use more main memory.

CMP systems can have shared caches. Essentially, having a shared L2 cache allows the on chip memory of each processing element to grow based on runtime requirements. Cache coherency protocols can be processed on chip, *allowing multiple processing elements to share a single cache line and save a significant number of memory operations.* For example, the Intel Duo's Smart Cache does not need to write to main memory when multiple cores write to the same cache line. This advancement improves the potential benefits of cooperation amongst processes.

Similarly, to CMPs, Simultaneous multithreading (SMT) provides the ability of a single processing element to execute instructions from multiple independent instruction streams concurrently. However, because SMT systems do not have additional functional units over single processor designs, SMT's main performance improvement is to hide the latency of main memory when executing multiple threads. Many data mining workloads are latency bound, and as such can benefit from SMT. *Algorithm designers should intelligently schedule thread execution to share as many memory fetches as possible*, a process called thread co-scheduling.

4.3.2 Parallel Graph Mining Challenges

The two most challenging obstacles to efficient parallel graph mining are load imbalance and efficient utilization of the memory hierarchy. We outline these issues below.

4.3.3 Load Imbalance

A fundamental challenge in parallel graph mining is to maintain a balanced load on the system, such that each processing element is working at full capacity for the duration of the overall execution. This challenge is exacerbated in graph mining because the time to mine a task is not known *a priori*.

Task Queuing

To allow a system to allocate tasks to nodes, a queuing model is required. Broadly speaking, two models exist, static task allocation and dynamic task allocation. Under static allocation, tasks are assigned to nodes *a priori*. Because the number of tasks in pattern mining is unknown, one must create an assignment function which maps a task to a node. For example, a hashing function could be used to map DFScodes to a node number, or a simple round robin approach. Then each task, such as mining a frequent one edge, could be assigned to the appropriate processor. These static techniques invariably suffer performance costs. Consider a situation with 5 tasks and 4 processors. If the tasks all require the same mining time, speedup can be at most 2.5 fold, which is 62%. The histograms in Figure 4.1 illustrate the degree of imbalanced for real world data sets. A synthetic data set is shown on the left, and a real data set is shown on the right. In both cases, frequent-3 edges were used to partition the

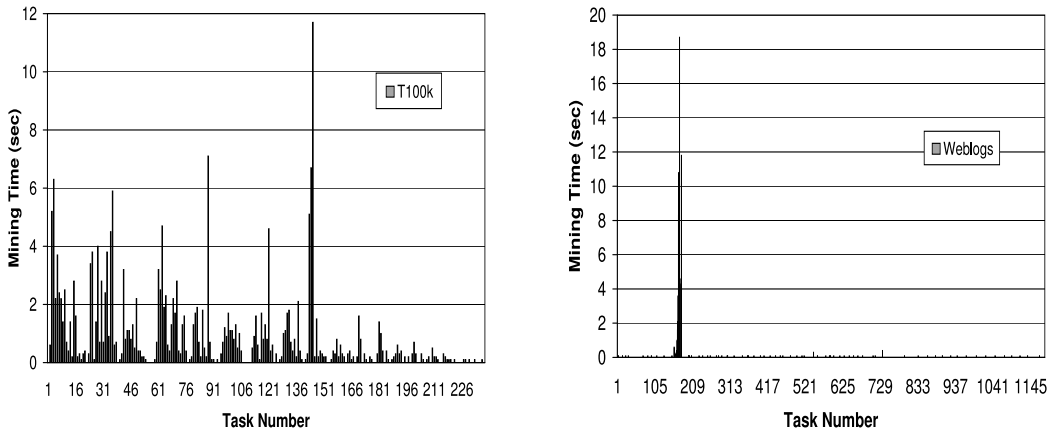


Figure 4.1: Task time histograms for level-3 partitioning with synthetic data (left) and real data (right).

mining phase. Only a few tasks in the real data set have appreciable mining times. We illustrate this phenomenon from a search space perspective in Figure 4.2.

This problem is understandable. In molecular data sets, for example, the edge C-C is much more frequent than He-C (this is an extreme example since He is a noble gas, however it illustrates the point). With static load balancing, it is conceivable that the two largest jobs in the mining could be assigned to the same node, affording almost no speedup. For this reason, we incorporate a queuing model to accommodate dynamic task allocation. Several queuing models are available; we describe them below. Each can be either First-In-First-Out (FIFO) or Last-In-First-Out (LIFO).

A *global* (or central) queuing model is a model in which each node adds and removes tasks from the same queue. A single mutex is required for all queuing and dequeuing. Contention is an obvious concern, because operations on the queue are

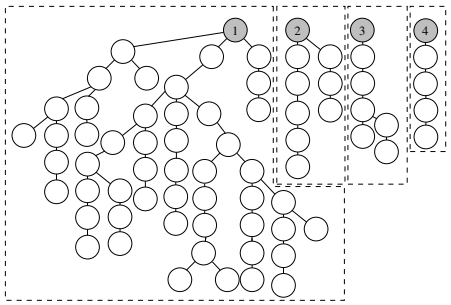


Figure 4.2: Typical graph mining workload.

serialized. One benefit is that there is maximum task sharing amongst nodes. Any task enqueued into the system is readily available to any idle node. Another benefit is ease of implementation. If a node finishes a task, and there are no other tasks in the global queue, it sleeps. When a node enqueues a task, it awakens all sleeping nodes. Termination occurs when a node finds no tasks in the queue and all other nodes are sleeping.

Hierarchical queuing is designed to allow task sharing between nodes. Each node has its own dedicated queue, which does not have a mutex. Nodes enqueue and dequeue from their own queue. If this queue is full, a node will enqueue into a hist queue. Conceptually, this queue is a level above the dedicated queues. This shared queue can either be above a subgroup of nodes, or over all nodes (i.e. easily extendable to more than two levels). If it is over a subset, then only that subset adds and removes from it. When a node's dedicated queue is empty, it attempts to dequeue from the shared queue. This shared queue has a mutex, and occurs some contention. If the shared queue is for a subset of nodes, then there will be another shared queue above it, which is for multiple node groups. This hierarchical structure ends with a queue with unlimited capacity at the top, which is shared by all queues. A node sleeps when

its queue and all queues above it are empty. If a node spills a task from a local queue to a global queue, it awakens the other nodes in the group. Termination occurs when a node finds no other tasks, and all other nodes are sleeping. The hierarchical model allows for task sharing, while affording fast enqueue and dequeue most of the time because the local queue has no lock.

In a *distributed* queuing model, there are exactly as many queues as nodes. Each queue is unlimited in capacity, has a mutex, and is assigned to a particular node. The default behavior for a node is to enqueue and dequeue using its own queue. Although this incurs a locking cost, there is generally no contention. If a node's queue is empty, it searches other queues in a round robin fashion, looking for work. If all queues are empty, it sleeps. When a node enqueues a task, it wakes all sleeping nodes. Before a node sleeps, it checks whether other nodes are sleeping. If all other nodes are sleeping, the algorithm terminates. The distributed model affords more task sharing than the hierarchical model, at the cost of increased locking. Also, as the size of the local queue decreases in the hierarchical model, it more closely resembles the distributed model.

Task Partitioning

Effective load balancing requires a task partitioning mechanism possessing sufficient granularity so as to allow each processing element to continue to perform useful work until the mining process is complete. In itemset mining, for large databases frequent-1 items generally suffice. This equates to the first loop in Algorithm 2, Line 5. However, for graphs this is not the case. In graph mining, a single frequent-1 item may contain 50% or more of the total execution cost. This is because task length is

largely dependent on the associativity in the dataset. We studied many real and synthetic data sets during the course of this work. Real data sets always showed more imbalance than synthetic data sets. We illustrate this imbalance with a depiction of the search space in Figure 4.2. Each circle represents a frequent subgraph to be mined (frequent-1 items colored gray), and each independent equivalence class has been boxed in with dashed lines.

4.3.4 Poor Memory System Utilization

Shared meta structures such as embedding lists can degrade memory system performance. An embedding list (abbreviated EL) is a list of mappings between a potentially frequent object and its locations in the database. The dependence on the structure is costly in a parallel setting, because all child graphs of the same parent must be mined before the parent's state can be freed. In addition, they consume significant memory, limiting the size of the problem which can be solved. An efficient algorithm must explore and optimize the trade off between no embeddings and full embeddings.

In graph mining data set accesses often lack temporal locality because the projected data set is typically quite large. These accesses also lack good spatial locality because most of the data structures employed are pointer-based. Locality issues are magnified on CMP systems because these architectures often have small caches. This is predicated by the real estate constraints of the chip, since for a fixed chip size each additional core will use silicon previously dedicated to cache. We performed a simple working set study to evaluate the benefits of improved dataset graph accesses. We compared two strategies for parallel graph mining. First, we forced threads to always

mine their own child tasks. Second, we allowed threads to mine other threads' tasks; that is, task stealing. The cache miss rates for the first strategy were, on average, three times lower than the second strategy. The reason is that when a task is stolen, typically most of the needed data set graphs are not in cache. As such, *algorithm designers should explore and optimize the tradeoffs between improved temporal locality and proper load balancing.*

4.3.5 Minimizing Extraneous Work

Given a balanced load and efficient memory system performance, the last major obstacle to an excellent parallel pattern mining algorithm is extraneous work. This extraneous work may come in the form of a) redundant computation, b) additional work due to parallelizing the serial algorithm, and c) queuing costs. We consider redundant work to be computation performed once in the serial algorithm but more than once in the parallel algorithm. We consider additional computation to be new computation which was not required in the serial setting, such as global aggregation operations.

Let W be the work performed by a serial algorithm. In system with n processors, a parallel implementation can at best do the same work, plus queuing costs q.

$$\Sigma_1^n w_i = W + q \tag{4.1}$$

This system would then be performing no extraneous work, except queuing costs. In graph mining, subgraph isomorphism can lead to mining the same graph twice. A simple solution is to verify the graph has not been previously mined. However, in a parallel setting, this would require synchronization and locking on the result set.

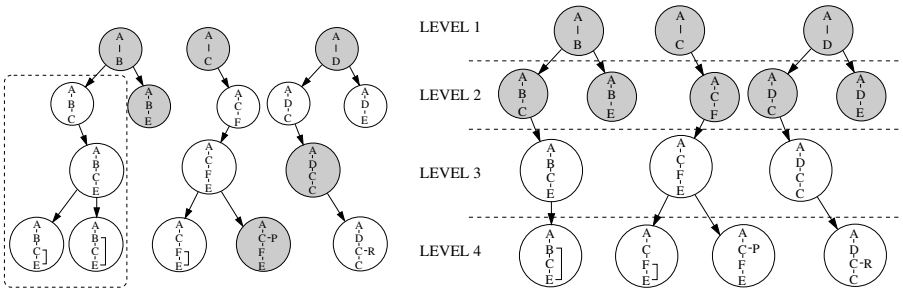


Figure 4.3: Adaptive vs. levelwise partitioning in graph data sets.

Algorithm designers should incorporate a canonical labeling system to remove the possibility of redundant mining without these synchronization costs.

4.4 Parallel Graph Mining Optimizations

We present our optimizations to improve graph mining performance on CMP architectures. Our goal is to minimize execution times provided a fixed number of computational units and an allotted memory budget.

Our pseudo code starts with Algorithm 4. It generally resembles the code from Algorithm 2, with the addition of lines 8-12 which insert frequent-1 edges into the queues. In line 1 the frequency of each edge is counted. Line 2 removes infrequent edges from the data set. Line 3 relabels the data set in frequency-ascending order. Lines 4-8 queue each frequent-1 edge into a processor's queue. The exact queue is chosen in a round-robin fashion. Although each frequent-1 item represents a separate task, these tasks are further partitioned as necessary. Line 9 generates POSIX threads. Lines 11-14 dequeue the tasks generated in line 7 for mining. The mining call occurs in line 13.

The mining process is presented as Algorithm 4. This algorithm proceeds as follows. In line 1 a canonical label evaluation occurs. If the DFScode is not the canonical representation for the represented graph, the function aborts mining. This check involves generating every possible labeling for the graph. If no label can be generated with a smaller sort order, the label is canonical. Line 4 adds the DFScode to the result set. Line 6 finds all candidate extensions for the DFScode by enumerating its location in every data set graph. Lines 7-22 generate recursive mining calls for each candidate extension which occurs at least σ times. Line 9 appends the candidate to the parent DFScode. Line 11 checks whether embedding lists are enabled and if the local queue is empty. If its not empty and embedding lists are enabled, the process recursively mines the task. Otherwise, lines 14 - 19 evaluate the case where embedding lists are not used. Line 15 again checks the state of the local queue. If its not empty, line 16 mines the task. If its empty, line 18 enqueues the task.

The enumeration process is provided as Algorithm 6. The process entails finding a mapping of the DFScode s in each of the possible data set graphs. If found, each possible extension is added to a child extension list C. Lines 2-4 use the mapping file, if available. Line 5 generates a new embedding structure if needed. Lines 7-9 finds all candidate extensions for each occurrence of s and adds them to C. Line 11 saves the embedding list if it is determined (in line 10) that they are in use.

Algorithm 4 ParallelMine

Input: *Global* Support threshold σ
Input: Graph Dataset D
Output: R = Set of frequent subgraphs

```
 1: Count the occurrences of all edge and node labels in D
 2: Remove infrequent edges and nodes
 3: Sort and relabel remaining nodes and edges
 4: for each Frequent-1 edge e ∈ D do
 5:    Initialize DFScode s with e
 6:    D_e = Graph IDs in D which contain e
 7:    Enqueue(D_e,s) to a random processors queue
 8: end for
 9: Spawn all threads
10: (* Each thread executes the following code *)
11: while all queues are not empty do
12:    D_e and s = Dequeue()
13:    ParallelSubMine(D_e,s)
14: end while
```

We consider two methods to control the granularity of these tasks, described below.

4.4.1 Adaptive Partitioning

Every call to Algorithm 5 generates a number of child candidates from which to extend the currently mined graph. Each candidate is then recursed upon (see Line 16). With *adaptive partitioning* [18], each processor makes a decision at runtime for each frequent child extension. A child is an edge tuple *(srcNode, destNode, srcLabel, edgeLabel, destLabel)* which can be added to the currently mined graph to create a new frequent subgraph. The child task may be mined by the creating processor, or enqueued, as seen in our 5 code on lines 16 and 17. Line 15 evaluates the boolean *canMine*, the first of our runtime hooks. The other hook, *canELMine* will be discussed

later in this Section. These hooks allow the algorithm to adapt to the properties of the system at runtime.

Algorithm 5 ParallelSubMine

```
Input: Graph Dataset D
Input: DFScode parent
Input: [Optional] EmbeddingList EL
 1: if parent is not the canonical label then
 2:    return
 3: end if
 4: R = R ∪ parent
 5: Children C = null
 6: Enumerate(D,C,s,EL)
 7: for each Child c in C do
 8:    if support(c) ≥ σ then
 9:       DFScode s = parent + c
10:       D_s = Graphs in D containing s
11:       if canELMine then
12:          ParallelSubMine(D_s,s,EL)
13:       else
14:          EL = null (*free memory*)
15:          if canMine then
16:             ParallelSubMine(D_s,s)
17:          else
18:             Enqueue(D_s,s)
19:          end if
20:       end if
21:    end if
22: end for
```

The value *canMine* is based on the current load of the system. We do not require a specific mechanism to make this decision, because in part the decision is based on the queuing mechanism used. In our experiments, we evaluate the size of the local thread's queue, using a threshold of 10% of the total number of frequent-1 edges. For example, with a global queue, it is natural to check the size of the queue. If it is

above a minimum value, then the creating processor mines the task without queuing; otherwise it enqueues it. Hierarchical models could set a threshold for any queue in its hierarchy. Another option could be to set a threshold for the smallest current size of any queue.

At each step in the depth first mining process, each subtask can be enqueued into the system for any other processor to mine. Thus the granularity of a task can be as small as a single call to 5, which is rather fine-grained. Adaptive partitioning is depicted in Figure 4.3 (left). In this example, task (A-B) had two children, namely (A-B-C) and (A-B-E). Tasks which were enqueued are shaded gray. A circle has been drawn around a task which was allowed to grow dynamically. Task (A-B-E) was enqueued, while task (A-B-C) was mined by the parent process. After mining (A-B-C), only one child was created, which was kept by the parent. The subsequent two children were both mined by the parent, because the queues had sufficient work so as to allow it. In practice we found that adaptive partitioning was the single most important optimization to lowering execution times.

Algorithm 6 Enumerate

Input: Dataset D
Input: Children C
Input: DFScode s
Input: [Optional] EmbeddingList EL
Output: C =all child extensions to s
Output: EL = a mapping of all occurrences of $s \in D$

```
1: for each Graph g ∈ D do
2:    if !(EL ==NULL) then
3:       Use EL to find s in g
4:    else
5:       Regenerate EL for s in g
6:    end if
7:    if s was found in g then
8:       add all children c in g to C
9:    end if
10:   if canELMine then
11:      Add Node to EL
12:   end if
13: end for
```

An alternative to adaptive partitioning is levelwise partitioning. We evaluate levelwise partitioning to provide a comparison to our approach. Levelwise partitioning deterministically enqueues tasks to a parameterized depth in the recursion tree. For example, in level-one partitioning, each frequent one edge is enqueued. These tasks are then mined to completion by the dequeuing processor. In level-two partitioning, a task is made for each child of a frequent one edge, and then each task is enqueued. Level-two tasks are then mined to completion by the dequeuing processor. Level-n partitioning implies level-(n-1) partitioning. If the work is balanced in the system, having the creator of a task mine the child will always be more efficient than enqueing the task, due to the benefits of affinity [74]. The child is always located in a subset of the graphs of the parent, so when those graphs are scanned, the cache will benefit

from temporal locality. Figure 4.3 (right) illustrates level-wise partitioning, set to level-2. Again, tasks which were enqueued are shaded gray [4].

4.4.2 Memory System Performance

We have found that parallel calls to malloc() simply do not scale. In a shared environment, heap allocation and deallocation are handled serially, so as to avoid returning the same memory address to two concurrent requests. Most operations in data mining use data containers without a predetermined size, so they use heap space. For example, when searching for a graph in the database, it is not known apriori in how many graphs it will be embedded. This is not an uncommon problem in shared memory environments, and researchers have proposed solutions [9]. Our solution was to give each node its own stack of heap space for recursive calls. This heap can grow and shrink as needed, typically in blocks of 4 to 50MBs. Because the algorithm is a depth-first recursion, markers can be set for each level of recursion, so as to allow simple deallocation without a loss of usable memory. The pointer in the local stack maintains the next available address. At the start of a recursive call, a marker is set holding the current pointer position in the stack. Then, when returning from the recursive call, the pointer is returned to this marked position. We store a separate marker (pointer position) for each level of recursion. These markers are unique to each thread of execution.

The benefit to this system is two-fold. First, allocations from each local heap are not serialized among threads. Second, allocation and deallocation requires only a couple lines of code to execute, as opposed to many lines of code and system calls required by traditional mechanisms. We believe the strategy is generally applicable to

[4] As an optimization, each processor actually keeps one child task, and enqueues the others.

other workloads. Also, if a particular meta structure's needed lifetime is not directly computable, traditional memory allocation can be used–the two can coexist easily.

It is common that algorithms can trade increased memory usage for improved execution times. We seek to leverage this principle to improve runtimes for graph mining applications by maintaining embedding lists when it is inexpensive to do so. Essentially, if we can guarantee the miner of a task will also mine its children, and there is sufficient memory, our algorithm will use embedding lists to mine those children.

We illustrate our proposed embedding structure in Figure 4.4. At the top of the figure we show a data set of four graphs. The node labels *V1..V17* are merely to allow us to reference a particular node with a pointer in the diagram, they do not exist in practice. Below these graphs are three dashed boxes, representing the embedding lists for graph *(A)*, graph *(A-B)*, and graph *(A-B-C)*. Again, we use labels *S1..S15* only for reference purposes. GID represents the graphID of the embedding. NP is a node pointer to the associated node in the graph. RMP is the right-most path of the graph we are growing. Although the right-most path can be determined strictly from the currently mined DFScode, we maintain it for efficiency. SP is the pointer to the previous graph's embedding list. For example, graph *(A-B)* is found in dataset Graph 2, and appears at *S7*. When the graph is grown with child *C* to create subgraph *(A-B-C)*, the embedding *S12* is extended to *S7*.

By maintaining the embedding lists between recursive mining calls, the algorithm need not rediscover the entire parent graph in each data set graph. These embeddings may be stored directly in the parent graph as fragment edge tuples when possible, which is often the case. However, when a task is enqueued, we do not include the

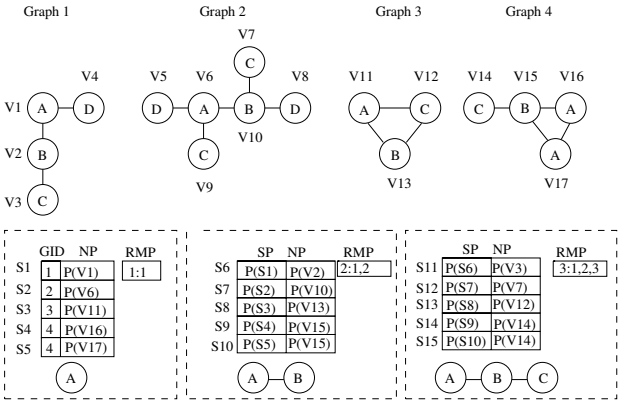


Figure 4.4: Example embedding list structure.

embedding lists. This allows us to free its associated memory, thus leveraging the local heap discussed earlier.

4.4.3 Minimizing Redundant Work

DFScodes afford the ability to verify the canonical representation of a graph. By using DFScodes, and partitioning the search space into independent tasks, there is no work performed in the parallel version which is not performed in the serial algorithm, except for queuing costs. Distributed queuing incurs a small cost for each queue access, since locks are required. However, adaptively allowing a task to grow to include its children tasks minimizes these costs. Tasks are only enqueued if there is insufficient work distribution to maintain full utilization of the processing elements.

4.5 Experimental Evaluation

We implement our algorithm in C using POSIX threads. We make use of four shared-memory machines. For CMP experiments, we use a dual processor machine consisting of two 2.33GHz quad core Intel Pentium Xeons with 4MB L2 cache and

Data set	# Graphs	# Node lbls	# Edge lbls
Weblogs	31,181	9,061	2
PTE	340	66	4
T100k	100,000	3700	1
NCI	237,748	37	3
nasa	1,000	342	1

Table 4.2: Data sets used for experiments.

8GB of RAM (totalling eight cores). For SMT experiments, we use a 2.80GHz Intel Hyperthreaded Pentium 4 with 1GB of RAM (two contexts). For testing scalability on more than eight processors, we use one of two SGI Altix systems–a 3000 with 64 GB of memory and 32 1.3GHz single core Itanium 2 processors, and a 350 with 64GB RAM and 16 1.4GHz single core Itanium 2 processors. All four machines run Red Hat Linux. Execution times do not include the time to read in the data set, which was not parallelized. In all cases, the graph database fits within main memory. The baseline for scalability is a single-threaded execution of our implementation.

We employ several real world data sets, shown in Table 4.2. PTE is a data set of molecules classified as carcinogens by the Predictive-Toxicology Evaluation project[5]. We also employ the complete NCI database[6]. Weblogs is a data set of web sessions, generated from web server request and response logs [89]. T100k is a synthetic data set made from the PAFI toolkit[7]. The nasa data set is a distribution of 1000 synthesized receivers for multicast trees[8].

[5]http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/PTE/

[6]http://cactus.nci.nih.gov/ncidb2/download.html

[7]http://www-users.cs.umn.edu/ karypis/pafi/

[8]http://www.nmsl.cs.ucsb.edu/mwalk/trees.html

data set	1	2	4	8
PTE 2%	55	27	13.8	7.0
Weblogs 0.07%	46	23	11.8	6.3
nasa 89%	104	67	46	24
NCI 0.4%	64	32	16.4	10.6

Table 4.3: Graph mining execution times (in seconds) for the dual quad core Xeon machine.

4.5.1 CMP Scalability

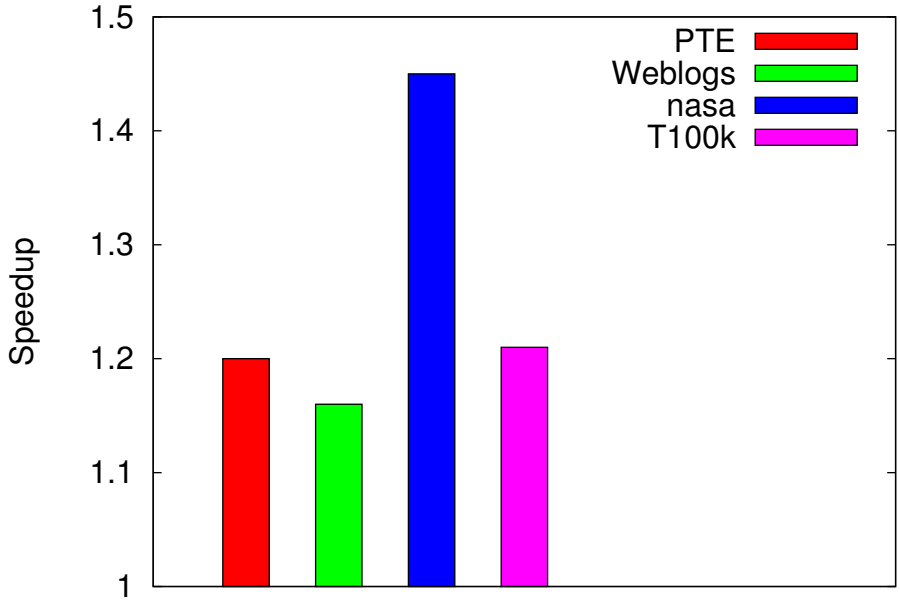
Execution times for several datasets executed on the dual quad core machine are given in Table 4.3. For the PTE and Weblogs data sets, execution times scale near linearly. For the NCI and nasa data sets, eight cores afford 4.33- and 6.0-fold speedups. Interestingly, execution times on the nasa data set scale far better on the Altix system. After inspection, we found that these datasets have larger working sets due to repetitive large tree patterns.

4.5.2 SMT Scalability

We briefly evaluate the benefit of SMT using hyperthreading. As seen in Figure 4.5, most datasets see only a modest improvement when compared to dual core execution time reductions. This is because the CPU utilization is high. However, as the working set size increases, hyperthreading shows execution time improvements. The nasa dataset illustrates this notion.

4.5.3 SMP Scalability

To measure overall scalability, we ran our algorithm on the Weblogs and PTE data sets. Speedups are ratios of the execution time for our serial implementation



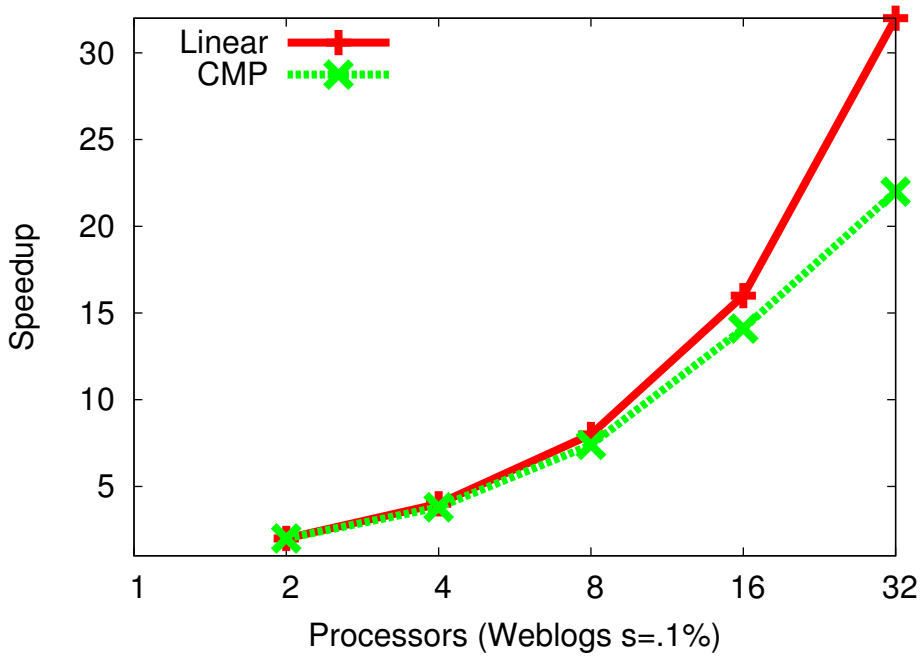
(a) SMT Scalability

Figure 4.5: Execution times on a two way SMT.

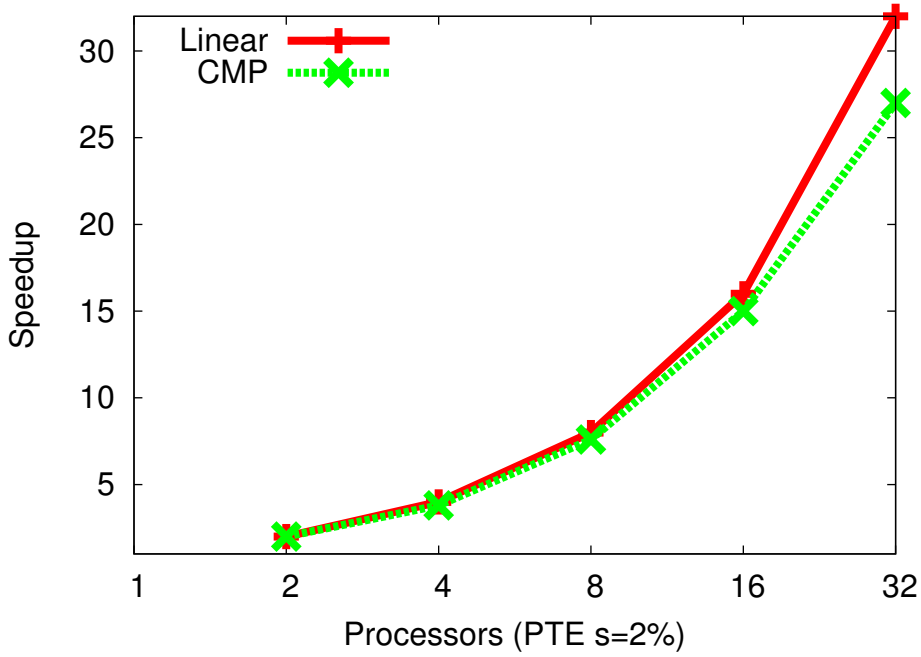
divided by our parallel implementation. All trials use the full data set. As can be seen in Figure 4.6, the algorithm scales near linearly as we increase the computational throughput of the system. These graphs include a linear speedup curve for reference. On 32 processors, the scalability ranges from 27 to 22.5 over these data sets. Weblogs has the lower scalability at 22.5. This can be attributed to its increased memory traffic. The frequent graphs in the Weblogs data set are larger than those in the other data sets, which requires the algorithm to traverse a larger percentage of each of the database graphs, deeper into the recursion. We note that scalability is generally maintained as support is reduced because there are more tasks, and those tasks can be mined by the parent node.

It is also clear that after 24 processors, the near linear speedup begins to degrade. This is primarily due to the nature of the size of a task. We believe that to afford linear speedup to more cores, future techniques for task parallelization should accomodate work sharing at an even finer granularity. Finally, we have found that our current strategies for parallelization do not sufficiently limit the working sets for individual threads. As the working sets increase in size, cache performance degrades, which then lowers end-to-end execution times as well as limits scalability.

To measure weak scalability, we partitioned the synthetic data set T100k into 32 equal chunks, and executed the algorithm with a number of chunks proportional to the number of nodes. Ideally, we would scale work precisely with increasing nodes. Graph mining runtimes are highly dependent on the associativity and distribution of the data, so we tuned support in each experiment to return the a graph result set proportional to the number of nodes. We use the synthetic data set T100k for this experiment because we believe its partitions more closely resembles the properties of the



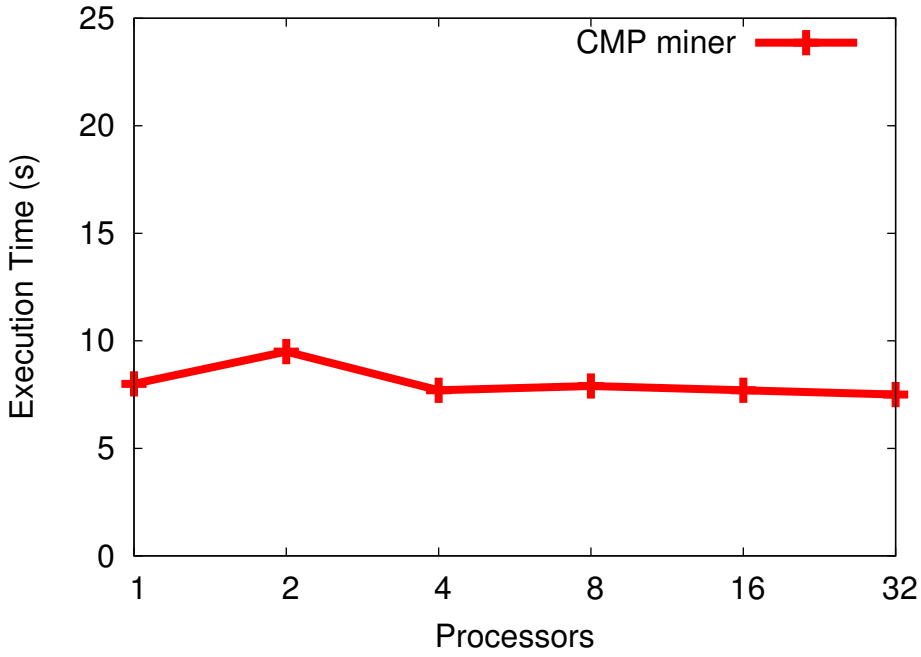
(a) Strong scalability on Weblogs



(b) Strong scalability on PTE

Figure 4.6: Strong scalability for CMP graph mining.

full data set. The results are presented in Figure 4.7. The algorithm accommodates the increase in data set size without a significant degradation in performance.



(a) Weak scalability on T100k

Figure 4.7: Weak scalability results for T100k.

4.5.4 Adaptive Partitioning Evaluation

To evaluate our task partitioning strategies, we employ data sets Weblogs, PTE, and T100k. T100k is used as a comparison to show that levelwise partitioning performs reasonably well with synthetic data (from PAFI). From the graphs in Figures 4.8 and 4.9, we see that adaptive partitioning outperforms levelwise partitioning.

For the Weblogs data set (Figure 4.9, top) adaptive partitioning affords a 22.5-fold speedup, where levelwise partitioning affords at most a 4-fold speedup. As the

number of partition levels increases from one to five, scalability generally increases, but it does not approach adaptive partitioning. Adaptive partitioning provides two key benefits. First, it provides an improved load balance in the system. Second, it affords the benefit of improved temporal locality, since it is more likely that a parent is mining its own children tasks.

Figure 4.8 (bottom) presents scalability for the T100k data set. Clearly levelwise partitioning has lessened the gap in speedup, but it still is only 15-fold vs 24-fold for adaptive partitioning. The improvement in levelwise partitioning is due to the balanced nature of the synthetic data set.

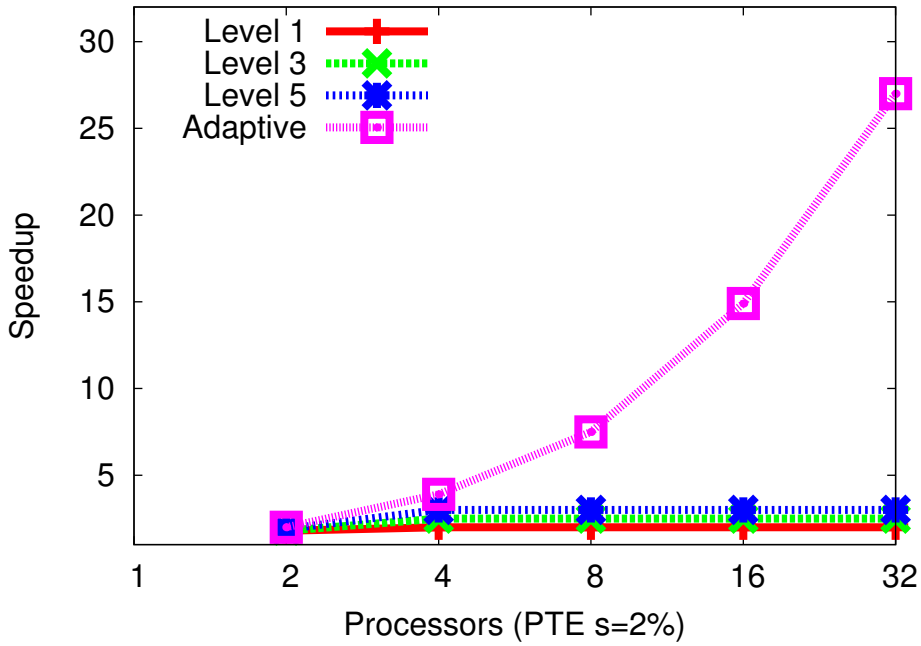
We simulated cache locality for dynamic vs. levelwise partitioning using cachegrind[9]. Figure 4.9 (bottom) illustrates the increased miss rates of levelwise partitioning. This same effect occurs if we co-schedule threads.

4.5.5 Queuing Model Evaluation

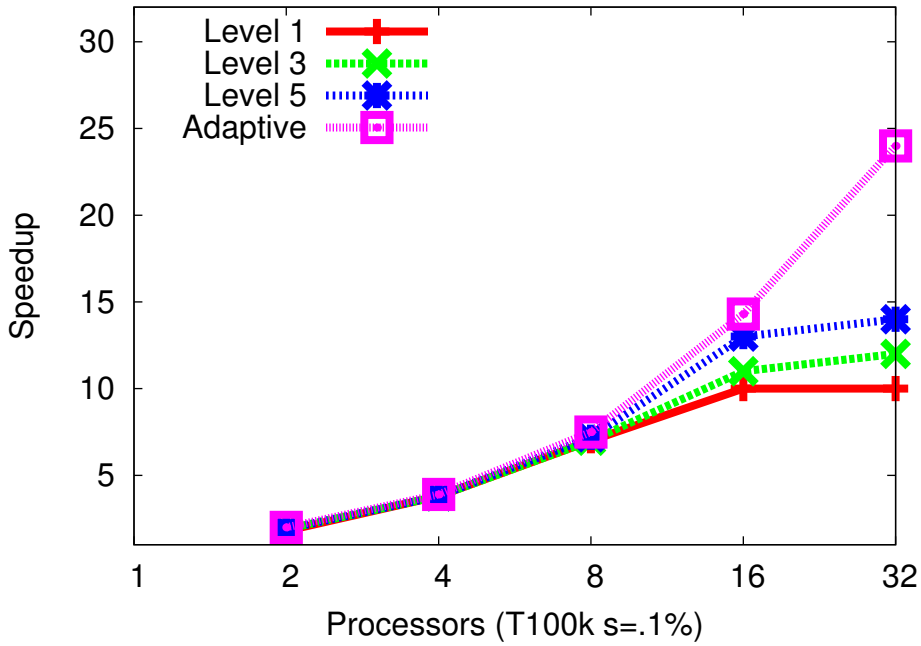
To evaluate the queuing models, we ran all three models on several data sets. We present the results of Weblogs here, in Figure 4.10. One global queue was not significantly worse than distributed queues. Contention on the global queue is not a major limiting factor in scalability at 32 nodes. Still, dynamic queues provide the best performance of the three. Hierarchical queues perform poorly, because task sharing is too low. As seen in Figure 4.1, the Weblogs data set has only a few frequent one edges with a significant number of child jobs. These jobs are not propagated to distant nodes in the hierarchical queue structure.

FIFO vs LIFO had no statistically significant impact. This is because once a node queues a task, it is unlikely to have high temporal locality when it is dequeued,

[9]http://valgrind.org/info/tools.html

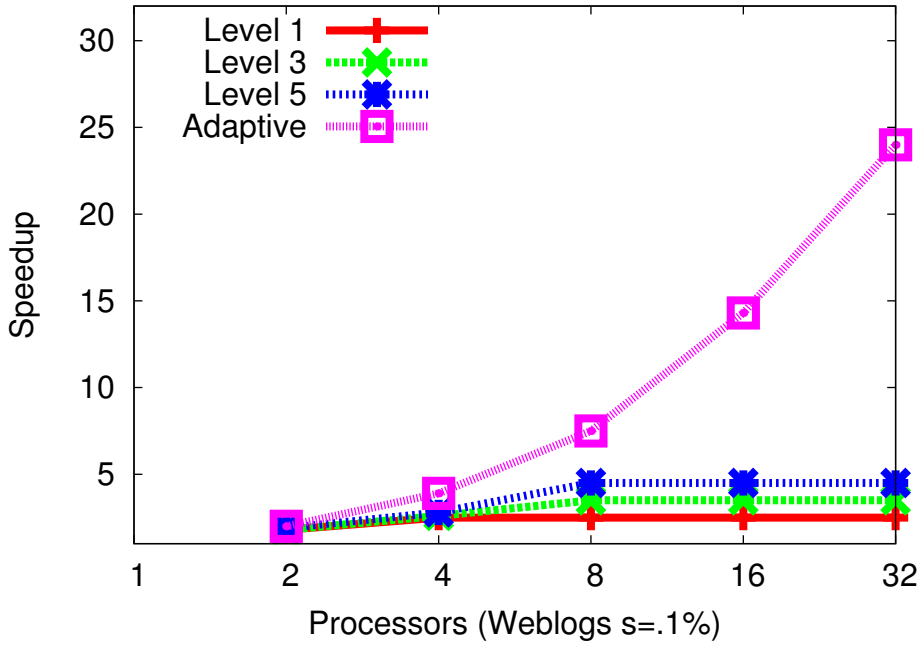


(a) PTE

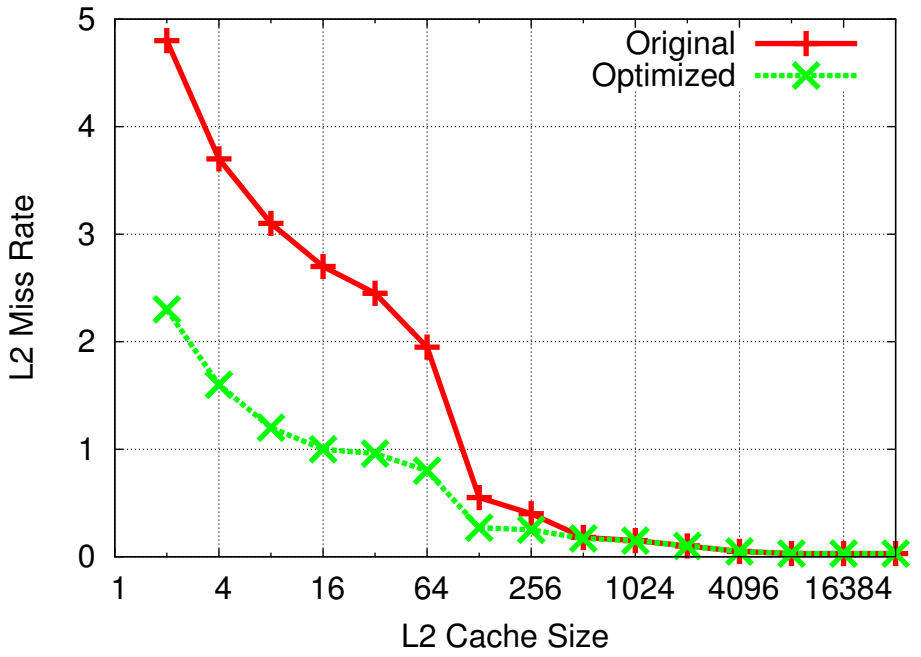


(b) T100k

Figure 4.8: Scalability of adaptive partitioning vs. levelwise partitioning for CMP graph mining.



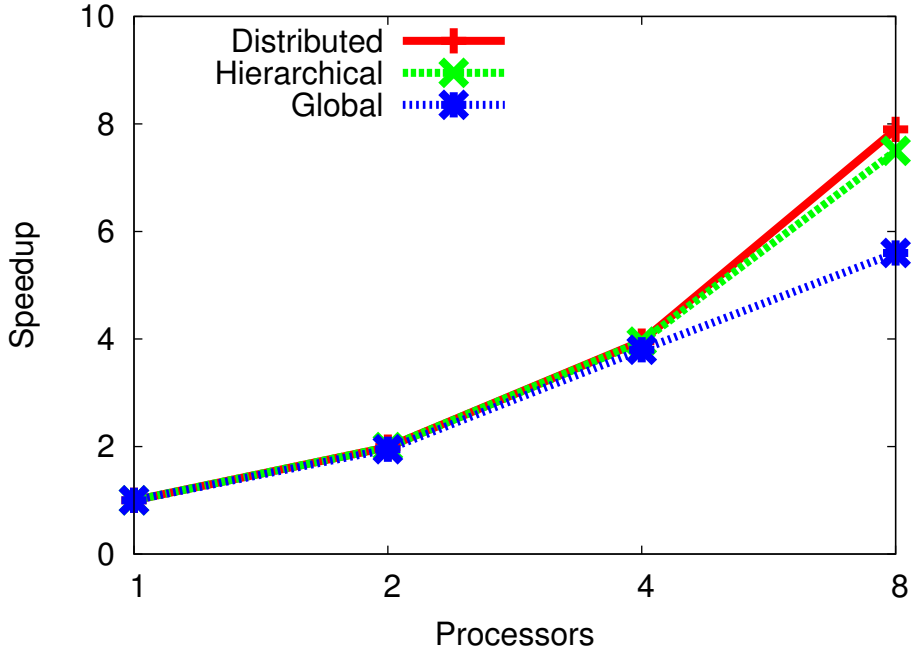
(a) Weblogs



(b) Weblogs

Figure 4.9: scalability of adaptive partitioning vs. levelwise partitioning for CMP graph mining (top), and working sets for dynamic vs. levelwise partitioning (bottom).

regardless of where it is located in the queue (top or bottom). Some data sets with low frequent one tasks benefit slightly with a FIFO queue because the children of the large tasks are more likely to be large as well, and the sooner they are distributed, the better the load balancing. Other data sets benefit slightly from LIFO because the most recently queued jobs have better temporal locality if they are dequeued within a few recursions from their queuing time.



(a) Queuing model performance

Figure 4.10: Queuing model scalabilities.

4.5.6 Memory Management Evaluation

We evaluated the benefit of our proposed thread level memory allocation strategy. The results for mining the PTE data set are shown in Figure 4.11. The figure

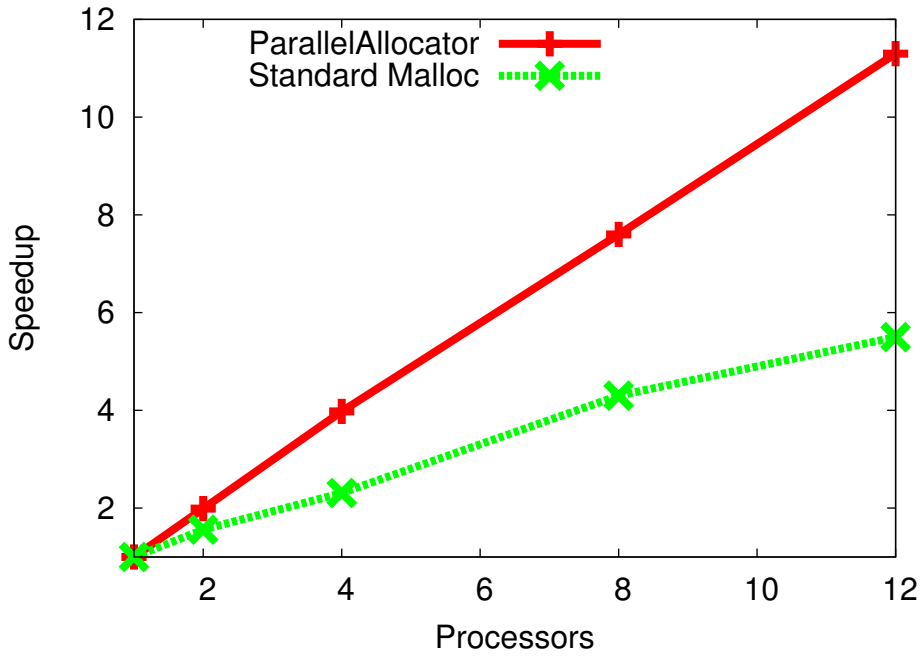


Figure 4.11: Improvements afforded by parallel heap allocation.

demonstrates that parallel allocation affords more than a 2-fold improvement, from 5.5-fold speedup to 11.3-fold speedup on 12 processors (using the PTE data set on the Altix 350 server).

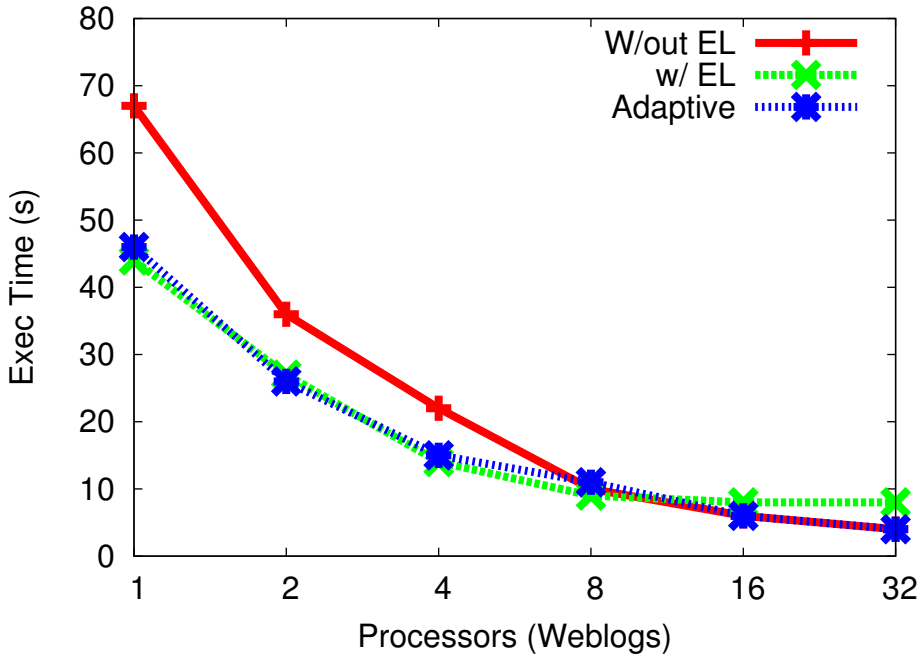
We evaluated the benefits of using dynamic embedding lists by mining the Weblogs and PTE data sets. We evaluate three different execution behaviors, namely never using embedding lists, always using embedding lists (to a static depth in the search space), and dynamically using embedding lists (based on the core count and main memory). The results are presented in Figure 4.12. We maintained a constant support value (as seen in the graph X axes). From the top figure, we can make several points. First, dynamic management results in improved performance over both static methods (either using or not using embedding lists). For example, adaptive state

management requires 38 seconds to mine Weblogs with one core, almost twice as fast as not employing state management. On 32 cores, it requires only 3 seconds, whereas statically using embedding lists requires 6 seconds. Never using embedding lists suffers when the number of processors is 4 or less, and always using embedding lists suffers when the number of cores is greater than 8. Second, using embedding lists in a static fashion results in the largest memory consumption. In Figure 4.12 (bottom), we compare our parallel code without embedding lists against a popular parallel graph miner, MOFA to mine the PTE data set. We can see that the former uses less than 10MB, while the latter uses 300MB when employing eight threads.

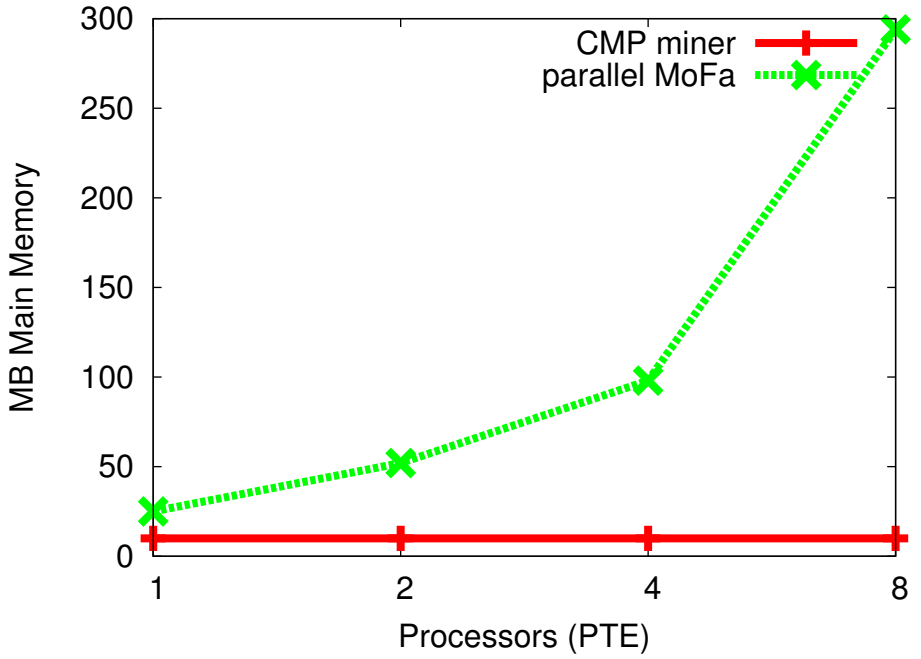
4.6 Discussion

It is clear that processor vendors now favor improving computational throughput by adding additional compute cores, as opposed to increasing the frequency of a single core chip. We believe this will soon require algorithm designers to address task parallelism. Data mining tasks often scale exponentially with increasing data set sizes, and as such could make excellent use of improved CPU cycles. However, as we have shown in this work, naive parallelization strategies often perform poorly on these systems. The limitations of the memory subsystem of CMP chips will be have a direct impact on parallel performance for most recursive algorithms which maintains state between successive calls.

We believe our proposed techniques are applicable outside the area of graph mining, since the issues presented herein are fundamental to efficient parallelization. For example, many algorithms stand to benefit from a reduction in overhead when allocating memory with concurrent threads. Also, the proposed adaptive partitioning



(a) Improvements with dynamic meta data



(b) Memory consumption with increasing threads

Figure 4.12: Memory system improvements.

and distributed queuing model could be applied to other domains with only modest changes.

4.7 Conclusion

We have shown that by allowing the state of the system to influence the behavior of the algorithm, we can greatly improve parallel graph mining performance. Specifically, our algorithm adapts by dynamically partitioning work when needed for improved load balancing, and dynamically managing memory usage. It uses distinct queues for each thread, as well as disjoint portions of heap space when possible. Through these techniques, we have developed a parallel frequent graph mining algorithm which can decrease mining runtimes by up to a factor of 27 on 32 processors. using far less main memory than other parallel graph mining strategies in the literature. We present these techniques in the context of emerging CMP architectures, where available main memory and its usage may be limiting factors in performance.

CHAPTER 5

PARALLEL DATA MINING ON THE CELL PROCESSOR

5.1 Introduction

As shown in Chapter 4, CMPs are a significant advancement in microprocessor design. In 2005, a joint venture by Sony, Toshiba and IBM (STI) produced a nine core architecture called the Cell BDEA. The layout of the Cell chip lies somewhere between other modern CMP chips and a high end GPU, since in some views the eight SPUs mimic pixel shader units. Unlike GPUs, however, the Cell can chain its processors in any order, or have them operate independently. And while most CMPs are MIMD with POSIX-style threading models, the Cell's eight cores are SIMD vector processors, and must get specialized task modules passed to them by the PPE. Target applications include medical imaging, physical model simulation, and consumer HDTV equipment. While the Cell chip is somewhat new, several data mining workloads seem quite amenable to its architecture. For example, high floating point workloads with streaming access patterns are of particular interest. These workloads could leverage the large floating point throughput. In addition, because their access pattern is known *a priori*, they can use software-managed caches for good bandwidth utilization.

This chapter seeks to map several important data mining tasks onto the Cell, namely clustering, classification and outlier detection. These are three fundamental data mining tasks, often used independently or as precursors to solve multi-step data mining challenges. All three tasks have efficient solutions which leverage distance computations. In addition, we will explore calculating the link-based PageRank [15] as well. PageRank employs streaming reads with mostly random writes. It will provide a use case which is less than optimal, so that we can explore the general-purpose nature of the Cell. In addition, it is an excellent representative of streaming graph computations, which have become increasingly pertinent since the emergence of the World Wide Web. Specifically, we seeks to make the following contributions to the computer science community. Our first goal is to pinpoint the bottlenecks in scalability and performance when mapping data mining workloads to this platform. We believe future streaming architectures could benefit from this study as well. We port these four tasks to the Cell, and present the reader with a detailed study regarding their performance. More importantly, our second goal is to answer the following higher level questions.

- *Can data mining applications leverage the Cell to efficiently process large data sets? Specifically, does the small local store prohibit mining large data sets?*
- *Will channel transfer time (bandwidth) limit scalability? If not, what is the greatest bottleneck?*
- *Which data mining workloads can leverage SIMD instructions for significant performance gains?*

- *What metrics can a programmer use to quickly gage whether an algorithm is amenable to the Cell?*

- *At what cost to programming development are these gains afforded?*

The outline of this chapter is as follows. A description of the Cell BDEA is given in Section 5.2. A background on the workloads in question is presented in Section 5.3. In Section 5.4, we present our Cell formulations of these workloads. We empirically evaluate these approaches, and discuss our findings in Sections 5.5 and 5.6. Finally, concluding remarks are presented in Section 5.7.

5.2 The Cell Broadband Engine

The *Cell Broadband Engine Architecture* [25] (Cell) was designed over a four year period primarily for the PlayStation 3 gaming console. The chip is also available in commercial blade servers through Mercury Computer Systems. Highly tauted, it was the winner of the Microprocessor Best Technology Award in 2004[10]. It is surmised by the high performance community that the Cell's commercial uses will allow it to be produced in sufficient quantities so as to support a low price tag. This thought, in conjunction with the Cell's significant floating point computational throughput and high off chip bandwidth, have led to discussion regarding utilizing the chip in large scale clusters.

The architecture features one general-purpose processing element called the Power Processing Element (PPE) and eight support processors called Synergistic Processing Elements (SPEs). It is depicted in Figure 5.1. The PPE is a two-way SMT multi-threaded Power 970 architecture compliant core, while all SPEs are single threaded.

[10]http://www.power.org/news/besttechaward.pdf

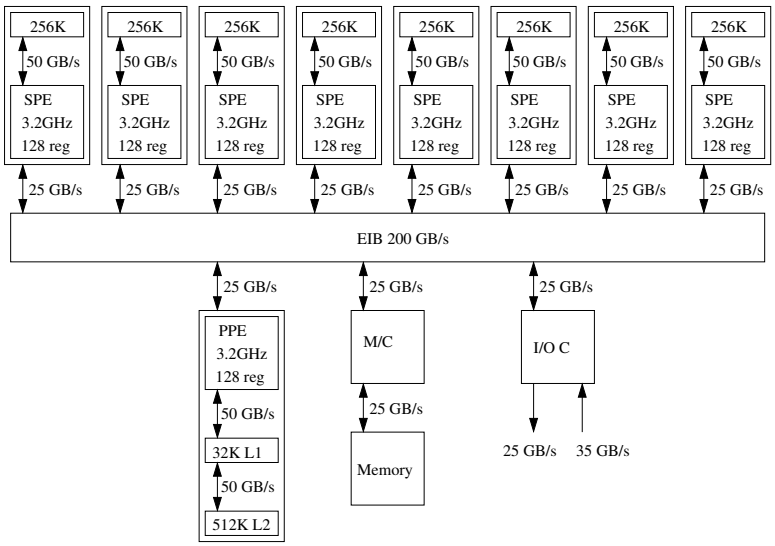


Figure 5.1: The layout and bandwidth of the Cell processor.

All processors are interconnected with a high-bandwidth ring network called the EIB. The Cell's design is one that favors bandwidth over latency, as the memory model does not include a hierarchical cache. In addition, it favors performance over programming simplicity. The memory model is software controlled. For example, all memory accesses must be performed by the programmer through DMA transfers calls, and the local cache at each SPE is managed explicitly by the programmer. Although this imparts a complexity on the programmer, it also affords the potential for very efficient bandwidth use, since each byte transferred is specifically requested by user software. There is no separate instruction cache; the program shares the local store with the data. Also, software memory management lowers required on-chip hardware requirements, thus lowering power consumption. At 3.2 GHz, an SPE uses only 4 watts per core; as a comparison, a 1.4GHz Itanium consumes about 130 watts.

Each SPE contains an SPU and an SPF. The SPF consists of a DMA (direct memory access) controller, and an MMU (memory management unit) to interact with the common interconnect bus (EIB). Bandwidth from an SPE to the EIB is about 25GB/s, both upstream and downstream (see Figure 5.1). SPEs are SIMD, capable of operating on eight 16 bit operands in one instruction (or four 32 bit operands). The floating point unit supports multiplication and subsequent accumulation in one instruction, (*spu_madd()*), which can be issued every cycle (with a latency of 6 cycles). This equates to about 25 GFLOPs per SPE, or 200 GFLOPs for eight SPEs, a number far greater than competing commodity processors. Each SPU has a low-latency mailbox channel to the PPU which can send one word (32 bits) at a time and receive four words. SPEs have no branch predictors. All branching is predicted statically at compile time, and a misprediction requires about 20 cycles to load the new instruction stream. Finally, the SPE supports in-order instruction issue only. Thus, the programmer must provide sufficient instruction level parallelism so that compile-time techniques can order instructions in a manner which minimizes pipeline stalls.

Algorithm 7 kMeans

Input: Dataset D
Input: k, the number of centers
Output: Each $d \in D \leftarrow$ closest center $c \in C$

```
1: while true do
2:    changed=0
3:    for each data point d_i ∈ D do
4:       assignedCenter = d_i.center
5:       for each center c_j ∈ C do
6:          d = dist(d_i,c_j)
7:          if d < d_i.Center then
8:             d_i.centerDistance = d
9:             d_i.center = j
10:         end if
11:      end for
12:      if !(d_i.center == assignedCenter) then
13:         changed++
14:      end if
15:   end for
16:   for each center c_j ∈ C do
17:      c_j = Mean of points i where c_i=j
18:   end for
19:   if changed==0 then
20:      break
21:   end if
22: end while
```

5.3 Data Mining Workloads

In this section, we briefly sketch the workloads under study.

5.3.1 Clustering

Clustering is a process by which data points are grouped together based on similarity. Objects assigned to the same group should have high similarity, and objects between groups should have low similarity. Clustering has many practical applications, such as species grouping in biology, grouping documents on the web, grouping

commodities in finance and grouping molecules in chemistry. It is considered an unsupervised learning algorithm, since user input is minimal and classes or group labels need not be determined *a priori*. There are many different mechanisms by which objects of a data set can be clustered, such as distance-based clustering, divisive clustering, agglomerative clustering, and probabilistic clustering. *kMeans* [72] is a widely popular distance-based clustering algorithm, and is the chosen algorithm for our study.

As its name implies, the *kMeans* algorithm uses the average of all the points assigned to a center to represent that center. The algorithm proceeds as follows. First, each data object is assigned to one of the k centers at random. In the second step, the centers are calculated by averaging the points assigned to them. Third, each point is checked against each center, to verify the point is assigned to the center closest to it. If any point required reassignment, the algorithm loops back to step two. The algorithm terminates when a scan of the data set yields no reassignments. A sketch is presented as Algorithm 7. Further details can be found elsewhere [72].

5.3.2 Classification

Classification is a common data mining task, whereby the label or class of a data object is predicted. For example, a classification algorithm may be used to predict whether a loan applicant should be *approved* or *rejected*. Classification is said to be *supervised* since classes are known and a training data set is provided, where each object in the set has been labeled with the appropriate class. There are many methods to predict labels. Examples include Bayesian networks, neural networks, nearest neighbors algorithms, and decision tree algorithms. Algorithm designers are faced

with several challenges when designing these solutions, including noise reduction, the curse of dimensionality, and scalability with increasing data set size.

One of the most widely used classifiers is the *k Nearest Neighbors* algorithm (kNN) [34]. kNN is said to work by analogy. A data object is classified by the most represented label of its k closest neighbors. In the worst case, the algorithm requires a distance calculation between each two data points. The method is considered *lazy* because a model is not built *a priori*; instead the training data is inspected only when a point is classified. In addition to avoiding model construction, kNN requires essentially no parameters and scales with data dimensionality. The data set is typically a collection of n-dimensional points, and the similarity measure is Euclidean distance. A sketch of the algorithm is presented as Algorithm 8. Further details can be found elsewhere [34].

5.3.3 Outlier Detection

Automatically detecting outliers in large data sets is a data mining task which has received significant attention in recent years [8, 24]. Outlier detection can be used to find network intrusions, system alarm conditions, physical simulation defects, noise in data, and many other anomalies. The premise is that most data fit well to a model, save a few points. These few points are then classified as outliers. As with classification, there are two common techniques, namely model-based approaches and distance-based methods. Model approaches build a model of the data, and then output data which does not fit the model. Distance-based approaches define a distance

calculation, and label points without nearby neighbors as outliers. Also like classification, distance-based detections schemes are well received because model construction is avoided, which is often a bottleneck with high-dimensional data.

Algorithm 8 kNearestNeighbors

Input: Dataset D
Input: k, number of neighbors
Output: $\forall d_i \in D, d_i.neighbors =$ closest k points to d_i

```
1: for each data point d_i ∈ D do
2:    d_i.neighbors = ∅
3:    for each data point d_j ∈ D where !(d_i == d_j) do
4:       dis = dist(d_i,d_j)
5:       if |d_i.neighbors| < k then
6:          d_i.neighbors = d_i.neighbors ∪ d_j
7:       else
8:          if max(d_i.neighbors) > dis then
9:             Remove farthest point in d_i.neighbors
10:            d_i.neighbors = d_i.neighbors ∪ d_j
11:         end if
12:      end if
13:      d_i.class = majority class in d_i.neighbors
14:   end for
15: end for
```

ORCA[8] is an efficient distance-based outlier detection algorithm developed by Bay and Schwabacher. It uses the nearest k neighbors as a basis for determining an outlier. The insight provided by ORCA is that once a point has k neighbors which are closer to it than the kth nearest neighbor of the weakest outlier, the point cannot be an outlier. Therefore, processing of the data point is terminated. To illustrate, consider the outliers in Table 5.1. This data represents the top 5 outliers, with the number of neighbors $k = 4$. Thus, an outlier is determined by the distance to his 4th neighbor. The weakest outlier is the outlier with the smallest 4th neighbor, in this

Neighbors →	1st	2nd	3rd	4th
Outlier 1	147.6	151.2	179.1	655.1
Outlier 2	342.2	387.5	409.9	458.2
Outlier 3	100.0	131.4	219.1	325.1
Outlier 4	87.2	89.8	107.3	210.0
Outlier 5	31.0	151.2	179.1	255.1

Table 5.1: An example set of outliers, where outlier 5 is the weakest.

case outlier 5. The threshold is then 255.1, since if any data point has four neighbors closer than 255.1, that point cannot be an outlier. Often times, k near neighbors can be found by scanning just a small percentage of the data set. A sketch of the algorithm is provided as Algorithm 9. Further details are available elsewhere [8].

For all distance-based workloads we implement the Euclidean distance as our similarity metric, which is a special case (p=2) of the Minkowski metric (given below).

$$d_2(x_i, x_j) = (\sum_{k=1}^{d}(x_{i,k} - x_{j,k})^2)^{1/2} \tag{5.1}$$

In practice, since the square root function maintains the total order on positive reals, most implementations do not take the square root of the distance. Based on our findings, we believe any distance calculation which touches every byte loaded will have similar results as those presented in Section 5.5.

Algorithm 9 ORCA

Input: Dataset D
Input: n, number of outliers
Input: k, number of neighbors
Output: O = top n outliers

```
1: O = ∅
2: Threshold = 0
3: for each data point d_i ∈ D do
4:    d_i.Neighbors = ∅
5:    for each data point d_j ∈ D where !(d_i == d_j) do
6:       d = dist(d_i,d_j);
7:       if d < max(O) or |d_i.Neighbors| < k then
8:          d_i.neighbors = d_i.neighbors ∪ d_j
9:       end if
10:      if |d_i.Neighbors| = k and max(d_i.Neighbors) > Threshold then
11:         break;
12:      end if
13:      if |O| < n then
14:         O = O ∪ d_i
15:      else
16:         if minDistance(O) then
17:            Remove weakest outlier from O
18:            O = O ∪ d_i
19:         end if
20:      end if
21:   end for
22:   Threshold = kth value from weakest(O).neighbor
23: end for
```

5.3.4 Link Analysis

PageRank [15] is a link analysis algorithm which assigns a score to each vertex in a graph based on the in-degree of that vertex. It was developed to assign weights to pages on the World Wide Web, such that pages with a higher PageRank constitute more useful information for a search engine. Essentially, it represents the probability that a random surfer will land on a particular page. The pagerank equation is provided as Equation 5.2, where T_i is the ith page linking into page A, $PR()$ is the PageRank

function, and $C(T_i)$ is the number of outlinks for page T_i.

$$PR(A) = (1-d) + d\left[\frac{PR(T_1)}{C(T_1)} + \cdots + \frac{PR(T_n)}{/C(T_n)}\right] \tag{5.2}$$

In matrix form, the problem consists of finding an eigenvector for the link matrix A with an eigen value of 1, where A is the n-by-n connectivity matrix of the web graph, that is, $a_{i,j}$ is 1 if there is a hyperlink from page i to page j and 0 otherwise. For larger matrices which do not fit in memory of a single machine, an adjacency list is used. In almost all practical scenarios, the power method is used to compute $x = Ax$, requiring about 75 iterations. In a distributed setting, each machine calculates its local PageRank vector for each outbound machine id, and then the values are interchanged in a synchronization phase. This simple two-step process continues until two successive iterations do not yield a change in a vertex's PageRank score.

PageRank presents an interesting challenge for the Cell processor because although the outlink lists can be read in a streaming fashion, updating the rank vector is essentially random access. The latter requires as much memory traffic as the former. Random access for the Cell is a significant penalty, requiring up to 1000 cycles per access even in light traffic conditions.

Algorithm 10 PageRank

Input: Graph $G = (V, E)$
Input: Vector M initialized to (V,1)
Input: Float *threshold*
Output: $M = (m_1 \cdots m_{|V|})$ where m_i is the PR of v_i

```
1: while d > threshold do
2:    for each v_i ∈ V do
3:       N_v+ = 0.15 * M_v
4:       E =outlinks from v
5:       for each outlink e_j ∈ E do
6:          N_e+ = 0.85 * N_v/|E|
7:       end for
8:    end for
9:    d = MAX(|m_i − n_i|)
10:   M = N
11:   N =(V,0)
12: end while
```

5.4 Algorithms

Developing algorithms for the proposed workloads on the Cell requires three components. First we must parallelize the workload. This is direct, as at least two of the three workloads are members of the *embarrassingly parallel* class of data mining algorithms. Second, we require an efficient data transfer mechanism to move data from from main memory to the local store. Third, we must restructure the algorithm to leverage the Single Instruction Multiple Data (SIMD) intrinsics available. In this section, we detail these components.

5.4.1 KMeans on the Cell

Parallelization of KMeans is straightforward. We partition the data set (which resides in main memory) such that two conditions hold. First, the number of records assigned to each processor is as balanced as possible. Second, the start boundary

of the each processor's segment is aligned on a 16 byte boundary [11]. This can be achieved by placing the first record on a 16 byte boundary, and then verifying that the number of records assigned to a processor satisfies the constraint below.

$$records * dim * sizeof(float) \ \% \ 16 == 0 \tag{5.3}$$

We simply assign the processor an even share of records, and add a record until it is properly aligned. If after adding a user-defined threshold of additional records, it is still not 16 byte aligned, then we pad it with the necessary bytes.

Efficient data transfer for *kMeans* is achieved by calculating a chunk size which results in landing on a record boundary, is a multiple of 16, and is about 6KB. At values below 6KB, the startup cost to retrieve the first byte is not sufficiently amortized. The maximum DMA call permitted by the Cell is 16KB, but we found smaller values afforded better load balancing opportunities. This chunk size can be calculated by simple doubling as shown in Algorithm 11. Line 1 calculates the initial chunk size, which is simply the size of a float times the number of dimensions of a point. Then lines 3-6 double the chunk size (in bytes) as well as the number of records to be fetched (line 5). When the chunk size in bytes exceeds 4096, the process terminates. It is then known that the chunk size is bounded below by 4096 and above by 8192. Also, as long as the number of records exceeds four, the chunk size will be a multiple of 16.

[11] The Cell's DMA controller requires 16-byte boundaries.

Algorithm 11 SPU GetChunksize

Input: M, the number of dimensions
Output: chunkSize is properly aligned

```
1: int chunkSize=sizeof(float)*M
2: int recordsToGet=1
3: while chunkSize < 4096 do
4:    chunkSize*=2;
5:    recordsToGet*=2;
6: end while
```

Restructuring *kMeans* to allow for SIMD distance calculations can be achieved by a) calculating the distance to multiple centers at once, b) calculating the distance between a center and multiple data points at once, and c) calculating multiple dimensions at once. We make use of two intrinsics, namely $v3 = spu\_sub(v1, v2)$ and $v4 = spu\_madd(v1, v2, v3)$. The former subtracts each element of vector $v1$ from $v2$ and stores the result in $v3$. The latter multiples the elements of $v1$ to $v2$, adds the result to $v3$ and stores it in $v4$. This second instruction effectively executes 8 floating point operations in a single instruction, with a 6 cycle latency. The latency can be avoided by calculating multiple points.

The strategy for calculating distances is shown in Algorithm 12. The input is a single record. Line 1 translate the record to vector form, to support vector intrinsics. Line 2 loops through each center, two centers at a time. Lines 3 and 4 translate the first two centers to vectors. Line 5 instantiates temporary vectors. Lines 6 - 11 loop through the dimensions of the data point, summing the distance in each dimension to the two centers. Lines 9 and 10 use the vector intrinsic *spu_madd*, which both multiplies four floats and adds the result in a single instruction. Thus, 16 floating point operations are performed by two instructions. It may be that the number

of dimensions in a point is not a multiple of four. Therefore, lines 12-19 perform the remaining calculations for these last few dimensions. Line 12 uses the intrinsic *builtin_expect()* to statically predict the branch. Lines 21 and 22 generate the final distance values by extracting the scalars from the vector temporary variables. These operations place the distance of the point to center one in variable *distance* and the distance to center two in variable *distance2*. Lines 23-30 then assign the point to the center if it has a smaller distance than the currently assigned center.

It is clear that the number of distance calculations in the inner loop can be expanded to improve throughput by further unrolling until each center in the chunk size is accommodated. In the case that the number of centers or dimensions is not modulo the largest desired block, a simple iterative halving flow of control is used to finish the calculation. The $if/then$ constructs after the looping are replaced by *spu_sel*() by the compiler, which removes simple branching. The SPUs send the number of reassigned data points back to the PPU through mailboxes. If any SPU reassigned a data point, the centers are recalculated and the SPUs are sent a message to perform another iteration; otherwise the SPUs are sent a message to terminate. The pseudo code for the $kMeans$ is shown in Algorithms 13 and 14.

Algorithm 13 is used by the PPU. Line 1 assigns a random center to each data point. Line 2 assigns the data to the number of available SPUs (subject to 16 byte boundaries). Line 3 spawns SPU threads. Lines 4-18 iterate assigning each point to the closest center. When the number of changed points is below a user-defined threshold (in the pseudo code, this value is set to zero), the program terminates. Lines 6-8 receive messages from the SPUs regarding the number of changed data points.

Lines 12-17 send messages to the SPUs. A value of zero is used to initiate termination (line 13) and a value of one is used to continue iterating (line 16).

Algorithm 14 is used by the SPU for *kMeans*. Line 1 gets the chunk size from the control block which is passed by the PPU during thread creation. This allows the SPU to move data in blocks from main memory to the local store in chunks that reside on record boundaries, which lowers branching and simplifies the code. Line 2 defines the number of bytes assigned to the SPU. Lines 4-19 assigns each point to its nearest center. Line 5 loads the centers into the local store (it is assumed that the centers completely fit within the local store). Line 7 loads a number of data points into the local store (per the chunk size). Lines 9-14 assign the nearest center to each point in the chunk. Line 11 calls the *AssignCenter* function. Line 13 increments the number of changed centers, if the nearest center is not the same as the nearest center from the previous iteration. Lines 17 and 18 generate and send the message to the PPU, which is the number of points which changed their center assignment.

An important issue when using the Cell is that any meta data which grows with the size of the data set cannot be stored locally. In the case of kMeans, the center assignment are an example of this type of data. The solution is to preallocate storage with each record when the data is read from disk. When each record is loaded from main memory to the SPU, the meta data is loaded as well, and when the record is purged from the SPU, the meta data is written to main memory with the record. This allows the algorithm to scale to large data sets.

Algorithm 12 AssignCenter

Input: Data *record* with M dimensions
Input: Centers C
Output: $record.center \leftarrow$ closest center $c \in C$

```
 1: vector v1 = (vector float*)record
 2: for i=0 to CenterCount step 2 do
 3:    vector v2=(vector float*)Center[i]
 4:    vector v3=(vector float*)Center[i+1]
 5:    vector float total,total2=0,0,0,0
 6:    for j = 0 to M/4 do
 7:       vector float res= spu_sub(v1[j],v2[j])
 8:       vector float res2= spu_sub(v1[j],v2[j])
 9:       total = spu_madd(res,res,total)
10:       total2 = spu_madd(res2,res2,total2)
11:    end for
12:    if _builtin_expect(!(M % 4 == 0),0) then
13:       int k = j
14:       for j = 0 to M%4 do
15:          float val1=(record[k*4+j]-center[i][k*4+j])
16:          total +=val1*val1
17:          float val2=(record[k*4+j]-center[i+1][k*4+j])
18:          total2 +=val2*val2
19:       end for
20:    end if
21:    float distance = spu_extract(total,0) + ... (total,4)
22:    float distance2 = spu_extract(total2,0) + ... (total2,4)
23:    if distance < record.centerDistance then
24:       record.center=i
25:       record.centerDistance=distance
26:    end if
27:    if distance2 < record.centerDistance then
28:       record.center=i
29:       record.centerDistance=distance2
30:    end if
31: end for
```

Algorithm 13 KMeans PPU

Input: Dataset D
Input: Number of Centers K
Output: Each $d \in D \leftarrow$ closest center $c \in C$

```
1: Assign each d ∈ D a random center
2: Partition D among P SPUs
3: Spawn P SPU Threads
4: while true do
5:    int Changed=0;
6:    for each processor p do
7:       Changed += p.mailbox
8:    end for
9:    for each center c_j ∈ C do
10:      c_j = Mean of points i where c_i = j
11:   end for
12:   if Changed==0 then
13:      ∀p ∈ P, p.mailbox ← 0
14:      break;
15:   else
16:      ∀p ∈ P, p.mailbox ← 1
17:   end if
18: end while
```

Algorithm 14 KMeans SPU

Input: Dataset Starting Address A
Input: Number of data points $totalData$
Input: Number of dimensions M
Input: Number of Centers K
Output: Each $d \in D \leftarrow$ closest center $c \in C$

```
1: GetData(D_p, I, K)
2: totalData = |D_p|
3: message=1
4: while message==1 do
5:    Load centers C into local store via DMA call(s)
6:    while totalData > 0 do
7:       Load data D_a into local store via DMA call
8:       totalData = totalData - recordsToGet
9:       for each data point d_j ∈ D_a do
10:         assignedCenter = d_i.Center
11:         AssignCenter(d_j,C)
12:         if d_i.Center != assignedCenter then
13:            Changed++;
14:         end if
15:      end for
16:   end while
17:   p.mailBox ← changed
18:   message ← p.mailbox
19: end while
```

5.4.2 kNN on the Cell

The main difference in construction between kMeans and kNN is that two streams are required. The first stream is the test data set (the data to be labeled) and the second stream is the training data set (the prelabeled data). The same chunk size is used for both streams, and is calculated with Algorithm 11. However, the record size is the dimensionality of the data plus k, where k is the number of neighbors to store. This allocation allows the SPU to store the IDs of the neighbors with the record, and limit local meta data. This can be doubled if the user requires the actual distances

as well; otherwise only one array of size k is kept on the local store to maintain this information and is cleared after each data point completes. This is a fundamental point when data mining on the Cell, which is to say that meta data must be stored with the record, to allow the Cell's SPUs to process large data. Synchronization only occurs at the completion of the algorithm.

5.4.3 ORCA on the Cell

The $ORCA$ construction is also similar to that of $kMeans$. $ORCA$ presents an additional challenge, however, because the effectiveness of computation pruning is a function of the threshold value. Without effective pruning, the algorithm grows in average case complexity from O($nlgn$) to O(n^2).)As the threshold increases, more pruning occurs. Partitioning the data set evenly may result in an uneven outlier distribution among the SPUs, thus the computation time per SPU becomes unbalanced. We can correct this by sharing local outliers between SPUs periodically. The strategy is to synchronize often early in the computation, and less frequently later in the computation. In the early stages, each data point has a higher probability to increase the threshold, since the set of outliers is incomplete. Recall that the threshold is the neighbor in the weakest outlier with the greatest distance. With all the SPUs maintaining separate outlier tables, their thresholds will vary. In most cases the thresholds will all be different, with the largest threshold being the best pruner. However, if all the SPUs share their data, the new threshold is most likely larger than any single SPU's current threshold. This is because the top five outliers from all the sets of outliers (one from each SPU) are the true outliers. Therefore, frequent synchronization early in the computation will support sifting these outliers to the top.

Partitioning the data set proceeds as it did for the previous two workloads. However, the chunkSize initially is set at the first record size satisfying Equation 5.3 greater than 512 bytes. Each successive data movement is increased, until a chunk size of about 4K is reached, which is optimal. As we will see in Section 5.5, chunk sizes larger than 4K result in a greater number of distance calculations.

At each synchronization point, each SPU writes its outliers to main memory. The synchronization is initiated by each SPU writing 1 to its mailbox to the PPU. When all SPus have written to their mailboxes, the PPU then takes the top n outliers from these eight sets and copies them back to main memory. When the SPUs start the next chunk, they also load the new outlier list, and with it the maximum threshold. When an SPU is finished with its portion of the data set, it writes a 0 to its mailbox. The algorithm terminates when all SPUs have written 0 to their mailboxes.

5.4.4 PageRank on the Cell

PageRank consists of three steps, repeated many times. First, the outlinks of a set of vertices are read into memory, along with their current PageRank values. Then, the values of the PageRanks for each outlink are incremented accordingly. Lastly, a convergence check is performed. For example, suppose vertex A has PageRank=10 and outlinks to vertices B,C and D. Also, let d in Equation 5.2 be 0.15. Then A will distribute 8.5 points, adding 2.83 to the PageRanks of B,C and D. The algorithm requires two size $|V|$ vectors, to store the previous and next PageRank values.

The parallelization proceeds as follows. First, the input data set is logically partitioned such that each SPU is responsible for roughly the same number of outlinks (not vertices). This division of labor must also reside on a 16-byte boundary. We

can use one global PageRank vector, or a set of vectors (one for each SPU). To avoid locking costs, and because typically the number of nodes is only 1/30th the number of edges, we used a unique vector for each SPU (totaling nine vectors when including the previous PageRank values). This requires an additional aggregation step, where each SPU is given a vertical portion of the vectors to sum. Each SPU PageRank vector must be updated many times per iteration for each SPU, since the local store can only hold a small portion of the vector at any given time. We investigated two methods to accomplish incremental vector updates. Note that these updates are to disjoint (non-contiguous) addresses in main memory. The first method used a sorted DMA list to write multiple values in one DMA call. The second method made a unique DMA transfer for each outlink list updated. Surprisingly, the latter proved much faster in practice than the latter (almost 20-fold faster). The reason is that the number of operations added to the compute phase to maintain the sorted vector far exceeded the marginal gain in channel stall time. In both cases, the current values must be loaded before being updated, since they are not present in the local store.

5.5 Evaluation

In this section we present a detailed evaluation of the proposed workloads and their optimizations on the Cell processor.

5.5.1 Experimental Setup

We execute the programs on a Playstation3 gaming console with Fedora Core 5 (PPC) installed. The PS3 provides the programmer with only six SPUs, as one is unavailable (rumored to be for improved yield) and another is dedicated to the game console's security features. It houses 256MB of main memory, of which about 175MB

is available. Performnace-level simulation data, such as cycle counts, was provided by the IBM Full System Simulator (Mambo), available in the IBM Cell SDK [12]. All data was synthetically generated (32 bit floats), so that we could manipulate parameters to evaluate the algorithms.

As a comparison, we provide execution times for other processors as well. In any work which attempts to evaluate an article (in this case, a processor) against others, implementation issues will arise[13]. Therefore, a few notes on these implementations. First, the other multicore implementations are that for Intel's PentiumD processor, which has two processing cores and the Xeon quad core processor, which has four cores. All other implementations were on single chip processors and use only one thread. Compiler flags had a large impact on performance, which is a topic in its own right. These implementations were compiled with a variety of different flags, and the best performing binaries are reported. For example, the Itanium performed best with *icc -fast*, and for the Itanium processor, the best performance was found with *icc -xW* which vectorized the distance calculation loop. In interesting cases, we provide two runtimes, one for Intel's compiler (icc) and another for the pulbic gcc compiler (at least -O3 flag). In all the trials for this section, we do not allow the PPU to perform useful computation, other than to act as the coordinator during synchronization. We experimented with providing the PPU with work, which in principle is comparable to adding another SPU. However, if the PPU requires more time in any one phase than the SPUs (and unlike the SPUs, the cache performance of the PPU is nondeterministic

[12]http://www-128.ibm.com/developerworks/power/cell/

[13]I was hoping to procure a labmate to implement the other codes, but found myself at the bottom of the laboratory hierarchy.

at compile time), then all 6 (or 8) SPUs are idle for that timeframe. Therefore, we choose to use the PPU for synchronization only.

The columns of Tables 5.4, 5.5 and 5.6 are as follows. The first four columns (five for kNN and $ORCA$) are input parameters, as shown in the headings. Each data point is an array of 32 bit floats. Columns 5-10 are the cycle statistics of the SPUs as a results of executing the program on the IBM simulator. Column 5 displays the Cycles required Per Instruction (CPI). Column 6 shows the percentage of the time that a single instruction is issued. Recall that the Cell SPU has two pipelines. Column 7 shows the percentage of the cycles that an instruction is issued on both pipelines. If this column were 100%, then all other columns would be 0% and the effective CPI would be 0.5, which is optimal. Columns 8-10 display the reason that there is not 100% double issue. Thus, columns 6-10 should sum to 100%. Branch stalls are due to branch mispredictions. Dependency stalls are due to a variety of reasons, for these workloads the common case is to stall on FP6, the floating point unit. This typically suggests that an instruction is waiting on the result of the previous instruction. Another common case is to stall waiting on a load instruction, which requires six cycles and moves the data from the local store to a register. Channel stalls are cycles lost waiting on DMA calls to load data chunks to the local store. Finally, the last column represents real execution time on the PS3.

5.5.2 Instruction Mix

The instruction mixes for each workload are presented in Table 5.3. From our description of these workloads, it is clear that the distance calculation dominates execution times for the steaming workloads, which is expected. Recall that our data

Processor	**Watts**	**MHz**	**Cores**	L2 Cache	**Compiler**
Itanium 2 (g)	130	1400	1	256KB(3MB L3)	gcc
Itanium 2 (i)	130	1400	1	256KB(3MB L3)	icc
Opteron 250	89	2400	1	1MB	gcc
Pentium D 2	95	2800	2	2MB (split)	icc
Xeon Quad Core 5345	80	2333	4	4MB (2x2)	icc
Cell 6 SPU	24	3200	6	1.5MB (LS)	IBM SDK 2
Cell 8 SPU (sim)	32	3200	8	2MB (LS)	IBM SDK 2

Table 5.2: Processors used for the evaluation.

	Kmeans	**KNN**	**ORCA**	**PageRank**
FP	35%	34%	31%	4%
ALU	17%	13%	23%	23%
SHIFT	24%	26%	17%	39%
LD/ST	10%	11%	13%	13%
LOGICAL	11%	9%	6%	17%
BRANCH	3%	5%	9%	4%

Table 5.3: Instruction mixes for the Cell processor implementations.

sets for *kMeans, kNN* and *ORCA* are 32-bit floats, and the Cell executes on 128-bit registers. For many floating point operations, this equates to 4 flops per instruction. However, about 30% of our operations are *spu_madd*() instructions, which multiply and add four 32-bit values in a single instruction. Therefore, although only 35% of the instructions are floating point, in actuality this is closer to 65% of the effective operations in a non-vectorized implementation. Still, we are not saturating the floating point units of the SPU in any of the workloads. The number of data points, dimensions, etc is considered fairly high by most readers – for kMeans we use trial 9 from Table 5.4, for kNN we use trial 5 from Table 5.5 and for ORCA we use trial 7 from Table 5.6). PageRank is quite different from the distance-based streaming workloads. It has only one tenth the floating point operations.

ORCA has the largest number of branch instructions at 9%. This is primarily because the threshold may eliminate the need for a distance calculation for a given point, and force the loop to terminate prematurely. Both *ORCA* and *kNN* have more branching than *kMeans* because the nearest neighbors are stored in a sorted array, which inherrently adds branching. All three workloads have a significant amount of loads and stores, which are required to bring the data from the local store to a register. Load and store instructions have a six cycle latency (not accounting for the channel costs to bring data chunks into the local store).

5.5.3 kMeans

The cycle statistics for *kMeans* is presented in Table 5.4 for various parameters. We fixed all trials to execute 30 iterations, to ease in comparisons. Interestingly, each iteration has the exact same statistics, since the computation is fixed and the SPU's

mechanics are deterministic (no dynamic branch prediction, no cache effects, in-order issue, etc.).

From Table 5.4, we can see that only when the number of centers is very low is there any appreciable channel delay. Thus for *kMeans* it can be concluded that moving data to and from the local store is not the bottleneck. In fact, most of the slowdown with the first trials is not due to the channel, but because the number of dimensions is sufficiently low to stall the pipeline on loop boundaries. This can be addressed with vector pipelining, albeit painstakingly so. Also, it would likely require padding, depending on the dimensionality of the data. Rather than use the memory space (the PS3 only has 256MB) we chose to use looping. As seen, when the number of dimensions increases, the SIMD instructions can be issued in succession, improving CPI (and FLOPs). Our initial implementation did not use SIMD instructions, and the CPI was quite low. Since each floating point instruction performed only one operation, each loop in the distance calculation used many instructions, and the issue rate was high. After SIMD instructions were used, the CPI increased, but execution times lowered.

The scalability is healthy from 1 to 6 SPUs. For example, in the last two trials, one SPU required 13.97 seconds and 6 SPUs required 2.38 seconds, for a speedup of 5.86. This near 6-fold speedup when moving from 1 to 6 SPUs is consistent in the other trials as well. Varying data set size behaved as expected, namely that twice as many points required about twice as much time (given the number of centers was far smaller than the number of data points). A final point to mention is that CPI and other statistics was generally fixed for a set of input parameters, regardless of the number of SPUs used. This is because, as long as there are enough data points to

Trial	Input				Output						
	Ctrs	Dims	Data Pts	SPUs	CPI	% Sgl Issue	% Dbl Issue	% Brch Stalls	% Dep Stalls	%Chnl Stalls	Exec t(sec)
1	10	2	200000	1	2.00	32	9	16	40	0	1.17
2	10	2	200000	6	2.00	33	9	17	38	3	0.20
3	10	10	200000	1	1.32	40	18	18	20	0	2.18
4	10	10	200000	6	1.32	40	18	18	19	1	0.37
5	10	40	200000	1	1.21	42	20	14	24	0	3.49
6	10	40	200000	6	1.21	41	20	14	25	1	0.77
7	20	40	200000	1	1.17	43	21	13	23	0	4.87
8	20	40	200000	6	1.18	43	20	13	23	1	0.84
9	20	100	200000	1	1.16	44	21	7	28	0	8.91
10	20	100	200000	6	1.16	44	21	7	27	1	1.54
11	40	100	100000	1	1.05	49	23	5	23	0	6.98
12	40	100	100000	6	1.05	49	23	5	23	0	1.19
13	40	100	200000	1	1.05	49	23	5	23	0	13.9
14	40	100	200000	6	1.05	49	23	5	23	0	2.38
15	40	100	400000	1	1.05	48	23	5	24	0	28.1
16	40	100	400000	6	1.05	49	22	5	23	1	4.97

Table 5.4: Statistics for Kmeans on the Cell processor.

fill one DMA load, and the channel contention is low, the SPUs will be performing independently.

Figure 5.2 illustrates the performance advantage of the Cell executing *kMeans* as compared to other commodity processors. The top figure compares the Cell using eight SPUs against four other commodity processors, each of which are using only a single core. The bottom figure compares the Cell using eight SPUs against the PentiumD using two cores and the Xeon Quad Core using all four cores. The parameters were DataPoints=200K, Dimensions=60, and Centers=24. The second best performance was afforded by the quad core Xeon, which is also a CMP.

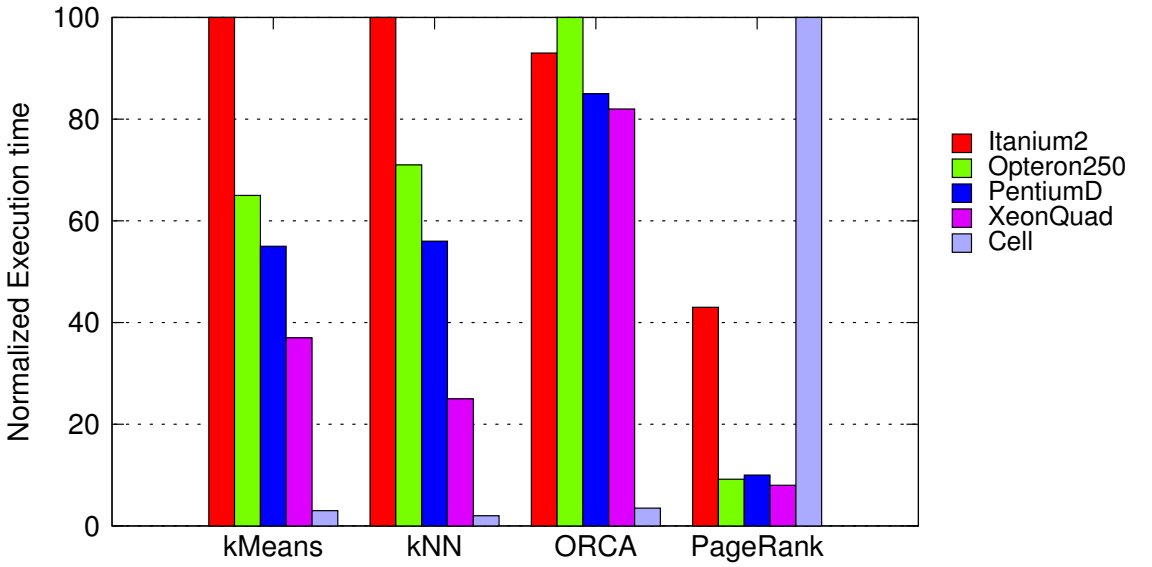
5.5.4 K Nearest Neighbors

The cycle statistics for *kNN* are provided in Table 5.5 for varying parameters. As with *kMeans*, *kNN* does not exhibit channel latency issues. Also, it can be seen that scalability is near linear. For example at 10 neighbors and 80 dimensions,

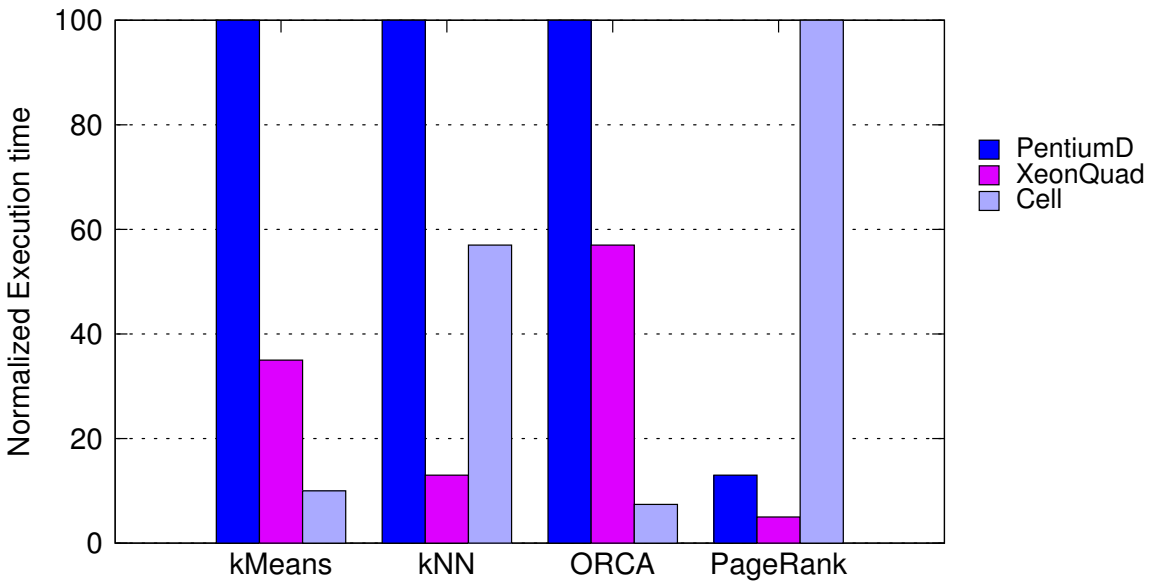
the execution time is reduced 5.95 times when moving from 1 SPU to 6. Also, the CPI is reduced from 1.53 to 1.02 when the workload rises from 10 neighbors and 12 dimensions to 10 neighbors and 80 dimensions. A larger number of dimensions results in longer record vectors, thus allowing more SIMD instructions per loop.

Increasing the number of neighbors degrades performance. This can be seen between the first trial and the third trial, where every parameter is held constant except for the number of neighbors, which is increased from 10 to 100. The subsequent CPI drops from 1.53 to 1.75, and branch stalls increase from 14% to 18%. Whenever a point d_j is found to be closer to the point being processed d_i, d_j must be added to d_i's neighbor list. This requires removing the weakest neighbor from the list and inserting d_j in sorted order. A larger neighbor list requires more search time, because a point is more likely to be a neighbor, and because adding that neighbor will be more costly. Recall that each (statically) mispredicted branch is a 20 cycle penalty.

Figure 5.2 illustrates the performance advantage of the Cell executing kNN as compared to other commodity processors. The top figure compares the Cell using eight SPUs against four commodity processors using a single core. The bottom figure compares the Cell using eight SPUs against the PentiumD using two cores and the Xeon Quad Core using all four cores. The parameters were TrainingPoints=20K, TestPoints=2K, Dimensions=24, and Neighbors=10. As was the case with $kMeans$, the Xeon's four execution cores afford it the second lowest execution times. It can also be seen that the Xeon scales linearly from one to four cores. Note the difference in single core execution times between the Xeon and the PentiumD – the Xeon has a lower clock frequency but executes in less time. It also requires less power and has twice the number of cores.



(a) Single Core Comparison



(b) Multicore Comparison

Figure 5.2: Execution time comparison between various processors.

Trial	Input					Output						
	Nbrs (k)	Dim	Train Pts	Test Pts	SPUs	CPI	% Sgl Issue	% Dbl Issue	% Brch Stalls	% Dep Stalls	%Chnl Stalls	Exec t(sec)
1	10	12	100000	1000	1	1.53	41	12	14	32	0	3.05
2	10	12	100000	1000	6	1.53	41	12	14	33	0	0.51
3	100	12	100000	1000	1	1.75	39	9	18	33	0	3.78
4	100	12	100000	1000	6	1.69	39	10	17	33	0	0.59
5	10	80	100000	1000	1	0.99	49	26	4	21	0	12.29
6	10	80	100000	1000	6	1.02	48	25	5	22	0	2.06
7	10	80	100000	2000	1	1.00	48	26	6	20	0	24.58
8	10	80	100000	2000	6	1.00	49	25	4	22	0	4.11
9	10	80	100000	5000	1	0.99	48	27	5	20	0	61.5
10	10	80	100000	5000	6	0.99	48	27	4	21	0	10.56

Table 5.5: Statistics for k Nearest Neighbors on the Cell processor.

Trial	Input					Output						
	Nbrs (k)	Dim	Otlrs	Data Pts	SPUs	CPI	% Sgl Issue	% Dbl Issue	% Brch Stalls	% Dep Stalls	%Chnl Stalls	Exec t(sec)
1	10	12	10	100000	1	1.40	49	11	12	28	0	15.8
2	10	12	10	100000	6	1.47	48	10	12	27	0	2.69
3	100	12	10	100000	1	1.96	35	8	24	22	0	65.31
4	100	12	10	100000	6	1.94	36	8	23	22	0	11.39
5	10	24	10	100000	1	1.36	53	10	10	27	0	15.36
6	10	24	10	100000	6	1.38	52	10	11	27	0	2.56
7	10	60	10	100000	1	1.28	59	9	6	26	0	19.36
8	10	60	10	100000	6	1.28	59	9	6	25	1	3.28
9	10	60	10	200000	1	1.27	59	9	6	25	0	48.26
10	10	60	10	200000	6	1.27	60	9	5	26	1	8.21
11	10	24	100	100000	1	1.36	53	10	10	27	0	28.2
12	10	24	100	100000	6	1.38	52	10	9	28	1	4.77

Table 5.6: Statistics for ORCA on the Cell processor.

5.5.5 ORCA

The cycle statistics for *ORCA*, collected from the simulator, are presented in Table 5.6. As with the previous two workloads, scalability from 1 to 6 SPUs is excellent. The CPI clearly drops when the number of neighbors increases, because we have more branch misprediction due to inserting and sorting into a longer neighbor list. Branch stalls increase from 12% to 24% when increasing the neighbor list from 10 to 100.This also occurred with kNN.

The algorithm handles increasing dimensions well, as seen in trials 2 and 8. The dimensions is increased from 12 to 60, but the execution time only increases 21%. The increased dimensions improve the CPI. While the CPI only drops from 1.47 to 1.28, we point out that each additional FP instruction executes approximately 6 operations, and these operations are only 31% of the workload. Doubling the data set size from 100K to 200K requires 2.5-fold longer running times. This not surprising, since the worst case performance of the underlying algorithm is O(n^2).

Figure 5.2 illustrates the performance advantage of the Cell executing *ORCA* as compared to other commodity processors. The top figure compares the Cell using eight SPUs against four commodity processors using a single core. The bottom figure compares the Cell using eight SPUs against the PentiumD using two cores and the Xeon Quad Core using all four cores. The parameters are DataPoints=200K Dimensions=32 Outliers=10, and Neighbor=40. The Xeon is again competetive. Still, the Cell is several times quicker than the others, at least 5.8 times faster than the Xeon. The Cell affords a better performance ratio when executing *ORCA* as opposed to *kMeans* and *kNN* because *ORCA* has working sets which exceed the available cache for the other commodity processors. The centers for *kMeans*, as well as the training set for *kNN* will fit in cache of the other processors. In these cases, locality of reference likely improves miss rates, thus improving the execution times for these competing technologies.

5.5.6 PageRank

The cycle statistics for *PageRank*, collected from the simulator, are presented in Table 5.7. Contrary to the previous three streaming workloads, scalability from 1 to

Trial	Input			Output						
	Graph	Data Pts	SPUs	CPI	% Sgl Issue	% Dbl Issue	% Brch Stalls	% Dep Stalls	%Chnl Stalls	Exec t(sec)
1	EU2005	862000	1	2.75	26.8	3.4	3.8	47.8	13.4	56.7
2	EU2005	862000	6	3.99	20.6	2.2	3.9	34.8	38	25.2

Table 5.7: Statistics for PageRank on the Cell processor.

6 SPUs is rather poor. This is attributed to i) the fact that a reduction operation is required for multiple processors in each iteration thus adding more work, and ii) there is significant contention for the memory bus. Channel stall times increase from 13% to 38%. The cycles required to write to main memory for the Cell is about 4 times slower than traditional processors.

Figure 5.2 illustrates the performance of the Cell executing *PageRank* as compared to other commodity processors. The top figure compares the Cell using eight SPUs against four commodity processors using a single core. The bottom figure compares the Cell using eight SPUs against the PentiumD using two cores and the Xeon Quad Core using all four cores. These trials were for 30 iterations of the publically available EU2005 data set (see Chapter 7 for further details). The Xeon is 22 times faster than the Cell for this workload. The poor performance of the Cell is due to the very frequent nonstreaming writes. The performance of the Cell is quite low when required to issue random writes to main memory. In particular for this workload, execution stalls on writes, since it may be necessary to rewrite to the same 16-byte block in the next instruction.

5.5.7 Streaming Channel Stalls

In light of the results for PageRank, specifically the degradation due to the memory subsystem performance, we now perform a study on the effects of channel stalls on one of the streaming algorithms. In this experiment, we vary the data transfer size from 64 bytes to 8192 bytes for *ORCA*, in an effort to gain insight on channel stalls on a real machine, since our earlier channel stall data was given by the simulator. All values for this experiment are taken from trials on the PS3, and all trials use six SPUs to maximize DMA contention.

Recall that the exact chunk size must be a) a multiple of 16 bytes, and b) on a record boundary. Thus for this experiment we let a record be four single precision floats. The size of the data set is 80,000 records, the number of centers, neighbors and outliers are 10. As can be seen in Table 5.8, very small chunk sizes degrade performance significantly. However, at 64 bytes, only approximately 50% of the slowdown is attributed to cycles waiting on the channel load to complete. The balance is due to a) the decrease in SIMD parallelization (only a few calculations can be vectorized at once) and the increased number of instructions to set up channel transfers. Although in our previous experiments transfer sizes were about 6K, this experiment shows that if they are sufficiently small, channel stalls can be rather costly.

The break in the curve occurs with transfers of at least 256 bytes. At this value, the number of DMA loads required to process the data set dropped from 175 million to only 9.6 million, and the cost of each transfer only increased from 1216 cycles to 1504 cycles. The end execution time dropped from 28 seconds to 4.31 seconds. The lowest execution time occurs at a transfer size of 4096 bytes. The reason is that transfers larger than 4096 have only a marginal improvement in transfer time per

Bytes	64	128	256	512	1024	2048	4096	8192
Execution time (sec)	28.2	13.7	4.31	2.78	1.82	1.45	1.20	1.40
DMA Loads (Millions)	175	78	29	9.6	3.3	1.0	0.36	0.10
Calculations (Millions)	216	218	222	226	228	243	264	303
Cycles per DMA Load	1216	1332	1483	1504	1950	2130	3960	7550
Channel wait time (sec)	11.1	5.01	1.89	0.64	0.26	0.11	0.07	0.04

Table 5.8: Channel costs as a function of data chunk size for 6 SPEs.

byte, but incur a significant increase in the number of computations made. Recall that each synchronization allows SPUs to share their largest outliers and threshold values. These synchronizations generally increase the average threshold at the SPUs and afford improved pruning. Also we note that the cycles stalls per byte continually decreases as the size of the transfer is increased.

5.6 Discussion

In this section, we revisit the questions posed in the Section 7.1.

Can data mining applications leverage the Cell to efficiently process large data sets? The answer for the streaming applications considered is yes. The small local store of the SPU did not pose a practical limitation, except for when executing PageRank. In most cases, only two buffers were needed, each of which was the size of an efficient data transfer (near 6KB). Only with *kMeans* did we use more than 15KB, since the full set of centers was kept on the SPU. Note that even here we could have avoided this extra buffer space – see for example our approach with *k-Nearest Neighbors*. Basically the N^2 comparisons among the centers and the loaded points amortizes the data loads to a sufficiently inexpensive cost. The overall insight is that in many cases any meta data associated with a record can

be inlined with that record, and moved to main memory when the record is ejected from the local store. For example, calculating K neighbors for D records requires $K * D * 4$ bytes. With low dimensional data, the number of records a DMA call can transfer becomes significant, and the local storage to maintain the K neighbors locally becomes the storage bottleneck. By simply inlining the K neighbors with the record, this storage requirement is mitigated (at the cost of lower cross-computational throughput per transfer). However, the local store size may be of greater concern with very large programs, since the instruction store and data store are shared. Clearly, for computations where the meta data exceeded local capacity, and was updated frequently (such as PageRank), the Cell does not afford good performance.

Will channel transfer time (bandwidth) limit scalability? If not, what is the greatest bottleneck? From our experience (not just limited to this study), workloads which touch every loaded byte and read and write in streams require less execution time on the Cell than on competing processors, regardless of the floating point computation requirements. For many data mining applications using distance calculations, this will be the case. Several studies, including one by Williams *et al* [107] recommend double buffering to reduce channel delays. For our streaming workloads the channel stall times were relatively nominal when compared to other issues, such as branch and dependency stalls. For example, from Table 5.4 we can see that in $kMeans$ only 10 centers and 2 dimensions was sufficient to reduce channel stalls to 3% of the cycle time.

As shown in Tables 5.4,5.6 and 5.5, branching is a significant bottleneck. From our experience, it is a penalty which can be difficult to avoid. Using *select*() instructions will only remove the simplest cases. Dependency stalls appear high as well, but these

costs can be lowered with additional loop unrolling and SIMD vectorization, and in fact were more than twice as high before we unrolled the outer loops. Branching is a natural programming concept and is used frequently. Eliminating branches, particularly when each flow of control requires complex operations, is not trivial and often cannot be amortized. For example, in our simple merge sort algorithm, up to 20was due to branching.

Which data mining workloads can leverage SIMD instructions for significant performance gains? Any streaming distance-based algorithm has the potential for significant gains. This work targets data mining workloads which are, in some senses, the best case for the Cell. An algorithm designer can leverage the Cell's high GFLOP throughput to churn the extensive floating point calculations of these workloads. Also, the predictable nature of the access patterns (namely streaming) allow for large data transfers, where each byte in the transfer will be used in an operation. In these situations, the Cell can be extremely efficient. In many trials, our results from the Cell's real execution times via the PS3 exhibit many-fold GFLOP improvements over the theoretical maximums for all other processors in this study. This is due to the 25+ GFLOPs afforded by *each* SPU.

What metrics can a programmer use to quickly gage whether an algorithm is amenable to the Cell? There are two questions to pose when evaluating the applicability of the Cell to workload. First, is the access pattern predictable and able to move data in large blocks? If so, then it is likely that chunks of data can be transferred to an SPU and most of those chunks will be involved in a computation. Second, is the workload parallelizable? In our experience, this second question is often easily answered. Data mining applications in particular exhibit significant

data-level parallelism, since it is common that each data object must be inspected. If either of these answers is no, then the Cell is not the best choice. For example, our experience with *PageRank* suggests that if a significant portion of the operations are non-streaming access to main memory, the end execution times will not be competitive with other commodity CMPs.

At what cost to programming development are these gains afforded? The learning curve is about six weeks. The programming model afforded by the Cell is somewhat more difficult than conventional CMPs, such as Intel's PentiumD processors. The programmer must explicitly move data to and from main memory as necessary. Also, implementation time is significantly higher on the Cell than typical processors since data movement is explicit. On average, we required about three times longer to code complicated routines on the Cell.

A benefit of choosing the Cell as a development platform is that the Mambo Simulator is quite useful when tuning implementations. It provides cycle-level accuracy for the SPUs, and allows one to step through assembly level executions a cycle at a time. The programmer clearly sees which instruction stall the pipelines. In this regard, while prototype-level programs are unnaturally difficult to implement, highly efficient implementations may in fact be easier. However, it is clear to us that an experienced programmer will be able to prototype an algorithm much faster on traditional CMPs than a Cell processor.

Several sources compare the Cell processor to GPGPUs. Both own multiple small processing elements which can be pipelined, both support SIMD instructions, and both require explicit data movement by the programmer. We did not find the pipelining capabilities to be a benefit for data mining workloads. This is likely due to the

fact that all data objects use the same kernel for processing. A second point is that if meta data did not fit in the space of a single SPU, it was also unlikely to fit in the space of 6 (or 8) SPUs.

It is clear that competing technologies such as Intel's line of Xeon processors are quickly closing the gap in both overall compute power and cycles per watt. In particular, the performance improvement from the PentiumD to the quad core Xeon is not trivial. It may be the case that without further improvements to the Cell processor, its window of opportunity will close. A 2-3 fold runtime improvement may not justify the added programming complexity for many application developers.

5.7 Conclusion

In this work, we develop algorithms and strategies for porting data mining applications to the Cell BDEA. Specifically, we illustrate that clustering, classification, and outlier detection leverage the available bandwidth and vector processing to experience execution time reductions, on data sets which fill main memory. In addition, we show that algorithms which frequently read or write to noncontiguous locations in main memory, such as PageRank, do not transfer well to the streaming architecture that has been incorporated into the Cell. Finally, we provide insight into the nature of a larger class of algorithms which can be executed efficiently on such a CMP platform.

CHAPTER 6

DISTRIBUTED PARALLEL ITEMSET MINING IN LOG-LINEAR TIME

6.1 Introduction

The emergence of increasingly larger data sets in many real-world applications has posed new challenges to researchers interested in data mining and information retrieval. Examples abound, including the Internet, scientific simulations, media repositories, and retail data. For example, conservative estimates place the size of web at *30 billion pages* and *500 billion hyperlinks*. In many cases, the rapid increase in data sizes has been spurred by continuing improvements in disk capacity technologies. Companies store data because there is no need to delete data, and with the hope of levering it for a business advantage sometime in the future. The challenge of mining large data is only expected to increase – commodity hard drives have increased in size ten-fold in the past seven years, and this trend is expected to continue. Given these large scale data stores, even algorithms that scale quadratically with the size of the data are impractical.

As discussed in Chapter 3, discovering frequent patterns is a common method used by researchers to process these large data stores. Frequent pattern mining, and

in particular frequent itemset mining, has been studied extensively by the research community [1, 52, 71, 117]. Researchers have identified avenues for algorithmic improvement. Examples of existing techniques include sampling to reduce data access costs[99], intelligent pruning of the search space[71, 52], and leveraging streaming techniques to reduce memory consumption[73]. In addition, researchers have also examined approaches for constraining the resulting set of patterns identified by such approaches. Important ideas here include algorithms that can identify maximal[21], closed[116], non-derivable[22] and profile patterns[113]. The reader can pose a simple question – *Is there a genuine need for another itemset mining algorithm?* We believe there is such a need, and now set to both prove this claim and offer a solution. Central to our argument that a new algorithm for itemset mining is needed we contend that all existing approaches suffer from several major limitations as noted below.

- First, the time complexity in the worst case is exponential with respect to the input label set [100]. For data sets with a large number of different input labels, computing all frequent itemsets at even moderately support values is cost-prohibitive.

- Second, extant methods by their very design emphasize the discovery of *globally frequent* patterns. Although frequency plays a role in interestingness, rare patterns within highly correlated localized subsets of the data are seldom uncovered by frequency-based methods.

- Third, and somewhat due to the first two problems, existing methods are often unable to uncover very long patterns because they typically enumerate the search space in increasing length. While recent research on the discovery of

colossal patterns [124] is a step in the right direction – at its core it still follows the "frequency" criteria and as such suffers from many of the limitations we note above.

We believe that the requirement for exact global support counting is at the root cause and argue for an alternative formulation. We build on some ideas proposed by Cohen *et al* [30] where the authors showed that high confidence association rules could be processed efficiently without using global support as a pruning mechanism. They proposed a k-way minimum hash to directly find interesting implication patterns. Although the particular solution presented in their work does not scale to more than two objects, it is our belief that there exist many use cases for itemset mining where the requirement for exact support counting can be relaxed. One such case study is presented as part of this article.

Specifically, we relax the definition of frequent pattern mining, moving toward the notion of parameter-free itemset mining. We do not require a minimum support or minimum confidence; instead we report interesting patterns occurring in at least two transactions of a localized subset of the data. Itemsets are reported based on a new metric for *itemset utility*. This metric is a function of the length of the itemset and its frequency but neither of these aspects are explicitly exploited by the proposed algorithm. We use a two phase algorithm to generate the patterns. In the first phase, database transactions are grouped based on the Jaccard similarity of their k-way min hash signature. We allow a heuristic lexicographic sort to bias the cluster, and iterate to balance the probabilities between distinct elements. Then, in the second phase, patterns are generated, starting from the longest patterns. In addition to the sequential algorithm, we present a multi-level parallelization of the same, targeting

emerging cloud cluster systems comprised of multi-core nodes interconnected via a high bandwidth, low latency network. Key features include:

- the algorithm first localizes the data into similar subsets so that locally interesting patterns can be found at low cost
- the algorithm finds long patterns with high utility, without enumerating first the sub patterns, or traversing the search space in a particular order,
- the algorithm has bounded computational and space complexity of $O(D \log D)$[14],
- the algorithm is shown empirically scale with dataset size, mining a 3 billion edge graph in 67 minutes on a single machine
- the algorithm parallelizes extremely well across a clustered system of multi-core nodes – near linear scalability on an eight core CMP node (up to 7.8-fold) and across a cluster of such nodes,
- the algorithm compresses transactional datasets very effectively (up to 3.5-fold) and can also be utilized for compressing extremely large (web-scale) graph datasets.

We evaluate the approach on a range of real datasets and compare and contrast with traditional itemset mining algorithms across several dimensions to demonstrate the utility and viability of the proposed approach. The rest of this chapter is organized as follows. In Section 6.2 we provide background information. Section 6.3 describes the proposed web graph compression benchmark, and its challenges. Section 6.4

[14] D is defined in Section 6.2

describes our itemset miner. We evaluate it in Section 6.5, present a discussion in Section 6.6, and then conclude.

6.2 Background

In this section, we discuss the frequent itemset mining challenge. As defined in Chapter 2, the goal is to enumerate all subset patterns in a database which appear in a minimum number of records (called the minimum support).

It may be that a user is already aware of the highly supported patterns. In these cases, the user typically lowers support, to discover less frequent but more interesting patterns. For many real world data sets, however, the number of patterns generated grows quickly as the minimum support is lowered. For example, the kosarak web data set [15] has 1,617 frequent items at 0.5% support, and 764,958 frequent items at 0.1% support. Because a human can only really investigate a small number of patterns, or even because the time to evaluate a them via automated means, researchers developed further metrics to prune the result set. *Closed* itemsets are frequent itemsets which have no superset of equal support. Although still a worst-case exponential operation, closed sets often take less time to compute than the full set of frequent itemsets. *Confidence* is defined as the support of the antecedent union'd with the consequent divided by the support of the antecedent, or if $A \rightarrow B$, then confidence is $support(A \cup B)/support(A)$. Confidence can prune the number of rules generated, but do not address the time required to generate the itemsets used for these rules.

[15] http://fimi.cs.helsinki.fi/

6.3 Problem Formulation and Case Study

In this article we explore an alternative formulation of the problem. We frame the discussion using data compression as a workload, and describe a hypothetical solution. We then informally disprove the optimality of the solution and offer a heuristic but efficient approximation.

We begin by defining the utility of an itemset or more generally a pattern as follows:

$$Utility = (F - 1) * (L - 1) \tag{6.1}$$

where F is the frequency of the pattern and L is the length of the pattern. This metric attempts to capture the *compression ability* of a pattern – essentially a pattern with higher utility can compress the transactional dataset more. Let D be a database of transactions $T_1 \ldots T_n$, where each transaction is comprised of a set of items drawn from the set $I = I_1 \ldots I_m$. Let us now consider the following compression scheme as part of our hypothetical solution. Let the itemset which maximizes utility be called the *optimal* itemset, g. To discover g, we enumerate the closed sets $c \in C$ of D at a support of 2, and for each set, calculate *Utility(c)*. Now let D' be D with each occurrence of g removed. Then, the optimal itemset of D' is a new itemset g', again maximizing the metric above, but for D'. We can continue to remove the optimal itemset, each time producing a smaller data set, until the data set is the empty set.

It is easy to show that for this problem – picking the optimal set of itemsets to compress need not necessarily be the top itemsets from the original data set D. In fact, even the optimal set g does not necessarily lead to optimal compression.

```
1: 1 2 3 4 5 6 7 8 9 10 11 12
2: 1 2 3 4 5 6 7 8 9 10 11 12
3:                  10 11 12
4:                  10 11 12
5:                  10 11 12
6:                  10 11 12
```

Figure 6.1: Data set for our counter-example.

Consider using *closed* sets to compress the data set in Figure 6.1. The maximum compression in the first step is to compress the first two transactions (reduction= $1 * 11 = 11$), resulting in a representation of 21. However, choosing the weaker 10-11-12 (reduction= $2 * 5 = 10$) results in a final representation of 20. Further, one can add additional 2-transaction patterns with similar properties to arbitrarily increase the gap between the optimal and achieved compression. In fact, it is the common case that when the chosen itemset is removed from D, the compression abilities of the remaining closed sets are no longer accurate.

These facts force us to abandon exact solutions and examine an approximate solution for compressing D. In particular, we must focus attention near the tail of the distribution, to be detailed next. Prior to that, we present a very practical case study for why such a problem formulation is interesting.

6.3.1 Web graph Compression

The web can be represented as a graph, with hyperlinks forming directed edges and web pages forming vertices. Link servers [11, 90] are designed to host the web graph and to support link structure queries. Example queries include PageRank [15], HITS [67], link spam detection [97], community discoverygibson1, and many others. Many queries are best executed as random access algorithms (e.g. shortest path) and

thus researchers seek to lift the entire graph into the RAM of a distributed cluster. However, as the web is massive in size, this is rather expensive. Considering that the web over 30 billion pages and 500 billion edges, a concise representation would require approximately one terabyte of RAM (500B*log(30 billion)/8 =2.5TB). Since many link servers store both inlinks and outlinks, the requirement grows to more than 5TB. Therefore, a compressed form of the web graph is interesting. We will investigate the challenge of web graph compression in greater detail in Chapter 7 – we introduce it here simply to provide a compelling use case.

Using the above utility-based problem formulation one can develop a mechanism to compress the web as follows. First we cast the adjacency list of the graph as a transaction dataset. Next we search for high utility frequent itemsets. Once found, each such itemset is replaced by a single item, and a new transaction is added, which holds the frequent set. The single added item to each transaction is the transaction-identifier of the new transaction. In essence, a bipartite directed clique is compressed into a new graph vertex, saving outlinks. Overall compression of the graph is then defined as the number of items in the original data set divided by the number of items in the compressed data set. To be precise, this denominator must include the items added as new transactions (as well as the additional item added to the compressed transaction), since without them there is clearly a loss in information.

Of course exploiting the discovery of high utility itemsets for web data poses a unique set of challenges in its own right. As noted above identifying an optimal set of itemsets to compress with is NP-hard. Even identifying the top-K utility itemsets is extremely expensive. Additionally, itemset miners typically do not store the location where the itemset was found. In the absence of transaction identifiers,

a linear sweep is required to perform the existence checks. This computation can be quite expensive. Therefore, an efficient algorithm will have to maintain the state of the source of the frequent itemset. The vertical representation may afford this for free, but this representation is better suited to small transactions. The web follows a power law, so many transactions are very long. Also, many vertical format miners move data between disk and memory often.

Many patterns of interest represent common structure in web page design, and will be present in small web sites. For example, many sites use a common navigation header (or left bar), which will likely be a frequent set for the outlinks of site's pages. However, many websites are only 10-50 pages in size. Therefore, mining at low support is of particular interest. Maintaining a highly scalable solution in the presence of the large number of transactions and unique labels at very low support is quite challenging, bringing to the surface the exponential nature of frequent set discovery. In the next section we discuss an approach that addresses these challenges.

6.4 Algorithm

In this section, we detail our algorithm for itemset mining. We call it *Localized Approximate Miner* , or *LAM*.

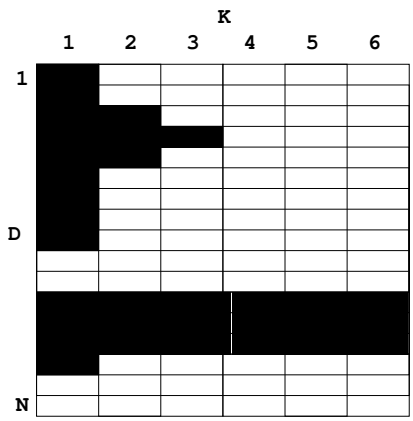


Figure 6.2: Clustering adjacency lists with probabilistic min hashing.

Algorithm 15 Localized Approximate Miner (LAM)

Input: A transactional data set D
Output: A Set of interesting itemsets S

```
1: for i=0 to NumberOfPasses do
2:    C = LocalizePhase(D)
3:    for all c ∈ C do
4:       Patterns P = MinePhase(c)
5:       Consume P
6:    end for
7: end for
```

The framework is provided as Algorithm 15. We name the grouping (or clustering) phase the *Localization* phase, which occurs on Line 2. It can be thought of as a probabilistic hierarchical clustering of the transactions; we seek to group transaction with similar items and similar overall length together. Then, each group is mined for patterns, in Line 4. Line 5 consumes the patterns dynamically, if required, since we maintain a pointer to the source of the pattern. We have chosen an iterative framework, so that results can be obtained quickly on very large data sets, and can

then be improved upon with successive iterations. We now detail scalable solutions for these two phases.

6.4.1 Phase 1: Localization

To maintain a scalable algorithm, we use probabilistic sampling on the data set. The goal is to group similar transactions together, where similarity is a function of the items in question, We restrict the computational complexity to $O(D \log D)$ to maintain scalability. We use K min-wise independent hash functions [16] to obtain a fixed length sample from each transaction, obtaining a $K * N$ matrix. We then sort the rows of the matrix lexicographically. Next we traverse the matrix column-wise, grouping rows with the same value. When the total number of rows drops below a user-provided threshold, or we reach the end of the hash matrix, we pass the transaction Ids associated with the rows to the mining process An example is depicted in Figure 6.2). Each row represents 10 transactions. In the first group of transactions, at $k = 3$ the 10 transactions represent one localized subspace, and would be mined together. In the bottom half of the figure, a second group is highlighted, consisting of 30 transactions whose all 6 hashes matched. This group represents another independent mining task.

For example, suppose in the first column there is a contiguous block of 200,000 rows with the same hash value. We then compare the second column hashes of these 200,000 rows. For each distinct hash value, we inspect the number of rows with that value. If cardinality is low, we call *MinePhase()*; otherwise we inspect the third column hash values for the selected rows, and so on. The lexicographic sort biases the sampling left-wise in the matrix, but multiple iterations afford a probabilistic

shuffling. This probabilistic sampling performs quite well in practice, grouping rows with high Jaccard coefficients.

Algorithm 16 LocalizePhase

Input: A Transaction Data set D
Input: Number of hashes K
Output: A set of local clusters C

```
 1: for all k ∈ Khashes do
 2:    for all T ∈ D do
 3:       Hash minH = HASH_MAX
 4:       for all i ∈ T do
 5:          h = hash(i, k)
 6:          minH = Min(h, minH)
 7:       end for
 8:       Add minH to matrix M for T
 9:    end for
10: end for
11: Sort M Lexicographically
12: pos=0
13: for all List N ∈ HashColumn(1) do
14:    Column col = 1;
15:    List N = GetList(N, pos)
16:    while |N| > threshold and col < K do
17:       col++
18:       List N = GetList(N, col, pos)
19:       C+ = N
20:    end while
21: end for
22: for all List N ∈ C do
23:    MiningPhase(D,N)
24: end for
```

Pseudo code for this first phase is provided as Algorithm 16. Lines 1 - 10 generate K minimum hashes for each transaction in D. Line 1 starts the outer loop, which iterates once for each hash. Line 2 starts the middle loop, which iterates once for each transaction. Line 3 sets the minimum hash value to the maximum value for an

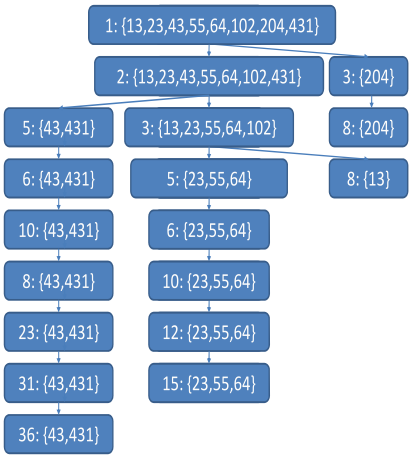


Figure 6.3: Example trie representation based on the data in Table 6.1.

unsigned int. Line 4 starts the inner loop, which iterates once for each item in the current transaction. Line 5 hashes the current item, using predefined primes based on the current hash iteration, k. Line 6 compares the hash to the current minimum. Line 8 adds the minimum hash to the hash matrix for the current transaction and hash column. Line 11 sorts the matrix lexicographically.

Lines 12-20 scan the hash lists to generate clusters of transactions. The first column serves to roughly group the transactions. Lines 16-20 scan the values in the first column. For each distinct value, the start row number and end row number is send to the *GetList()* (line 18). Line 16 prunes the list by comparing the current list with their next column hash values[16]. Line 19 adds the final list to the cluster set. Lines 22-24 then call the function *MiningPhase* for each list of transactions.

[16]In practice, this is implemented via recursion, such that every member of the set from list N is eventually passed to the mining process.

Trans Id	Items
23	6,10,5,12,15,1,2,3
102	1,2,3,20
55	2,3,10,12,1,5,6,15
204	1,7,8,9,3
13	1,2,3,8
64	1,2,3,5,6,10,12,15
43	1,2,5,10,22,31,8,23,36,6
431	1,2,5,10,21,31,67,8,23,36,6

Table 6.1: Example localized data set passed to the mining process.

6.4.2 Phase 2: Localized Approximate Mining

The goal of the mining phase is to locate long itemsets which optimize Equation 6.1. As discussed earlier, exact itemset mining is an exponential computation. Using a simple intersection of the provided links is too conservative, as many subpatterns are unnecessarily disregarded. Therefore, we instead focus on a greedy mining process which is bounded by $O(D \log D)$. The mechanism is to construct a trie of the transactions, and then to use long paths in the trie to construct patterns.

The algorithm proceeds as follows. An initial scan is performed to retrieve a histogram of the items in the transactions of the cluster, and then the transactions are reordered such that the most frequent labels sort first. Then each transaction is added to the trie. Each node in the trie has a label and a sorted set of transactions which contained the itemset prefix. Only items which occur at least twice are inserted into the trie. Then it is traversed to record patterns of interest.

For example, consider the data provided in Table 6.1. A total of eight transactions were provided. After removing singletons, the trie is constructed from the histogram in Figure 6.3, as shown in Figure 6.3.

Pattern	Node List	Uty.
1,2,3,5,6,10,12,15	23,55,64	14
1,2,5,6,10,8,23,31,36	43,431	8
1,2,3	13,23,55,64,102	8
1,2	12,23,43,55,64,102,431	6

Table 6.2: List of potential itemsets, where utility is defined as $(|P|-1)*(|NL|-1)$.

Id	1	2	3	5	6	10	8	12	15	23	31	36	7	9	20	21	22	67
Count	8	7	6	5	5	5	4	3	3	2	2	2	1	1	1	1	1	1

Table 6.3: Histogram for the example localized data set.

Algorithm 17 MinePhase

Input: A Transaction Data set D
Input: A list of transaction Ids N
Output: A modified list of transaction Ids N

1: **for all** $Id \in N$ **do**
2: **for all** $item \in n$ **do**
3: $Counts[item]++$
4: **end for**
5: **end for**
6: Sort $Counts$
7: Tree T
8: **for all** $Id \in N$ **do**
9: List L
10: **for all** $item \in n$ **do**
11: **if** $Counter[item] > 1$ **then**
12: $L+ = item$
13: **end if**
14: **end for**
15: Sort L using Counts
16: Add L to T
17: **end for**
18: List $PI = null$
19: GeneratePotentialItemsetList(PI,$T.Root$)

A walk of the trie is performed, and at each leaf with a transaction list length greater than one, a potential itemset is processed. The node is then colored as *processed*. From this node, a traversal to the root is performed. Whenever a parent node owns a longer transaction list, and it has not been colored, it is added to the list of potential itemsets. After the traversal completes, the potential list is sorted according to Equation 6.1. The list is then processed for itemsets. After each is processed, it is added to the data set, and then the associated transaction Ids in its path up the tree are deleted. It may be the case that subsequent potential itemsets are reduced in utility because the original ordering is greedy[17]. The actual utility is recomputed (in $O(1)$ time) before mining, and discarded if it is not fruitful. The potential itemset list for the running example are presented in Table 6.2.

Pseudo code for the mining process is provided as Algorithm 17. Lines 1 - 5 calculates a histogram of the items for all the transactions passed. Line 1 starts a loop which iterates once for each transaction Id in the list N. Line 2 starts a loop which iterates once for each item in the current transaction. Line 3 increments a counter for the current item. Line 6 sorts the counts in decreasing order, such that the most frequent item is in the first position.

Lines 7 - 16 examine each transaction, form a transaction of its items, and add it to the trie. Line 8 initiates a loop for each transaction Id. Line 9 instantiates a list L that stores the current transaction. Line 10 initiates a loop for each item in the current transaction. If the item appears more than once in N (line 11), then the item is added to L. Line 15 sorts L in descending frequency, based on the *Counts* array. The histogram ordering improves the overlap in the trie [52]. Line 16 adds L to the

[17]Subsequent sorting can be performed on the list but in practice we found this did not significantly change end to end compression values or execution time.

trie. Line 19 walks the tree to generate a list of potential itemsets. It is a function call to Algorithm 18, *GeneratePotentialItemsetList*.

GeneratePotentialItemsetList walks to the leaves of the trie (or to the last node in the path with a list length greater than one), and then walks back to the root. During the traversal back to the root, potential itemsets are created and added to a list. The pseudo code is provided as Algorithm 18. Line 1 initializes a variable, $Mark$, which stores whether or not the current trie node should be marked for adding an itemset. If the node has no children, it is a leaf and should be marked. If it has children, but at least one of the children does not have a count greater than one, then effectively, it is the leaf for that path and it should be marked. If a child has a transaction Id list size greater than one (checked in line 3), then the node is recursed upon (line 4). Adding a trie node as a potential itemset is accomplished by Algorithm 19.

Algorithm 19 starts at a leaf node in the trie and proceeds as follows. In Line 1 the transaction Id list size is stored. As long as the depth is greater than one, and the node has not been marked, then the pattern is added to the pattern list (line 2). Then, lines 6-9 walk up the path to the root, marking nodes along the way. At any point, if the Id list size increases, a recursive call is made.

6.4.3 Complexity Analysis

The complexity of the proposed approach is of particular interest, due to the size of the problem at hand. For a data set of size D items, the clustering phase is $O(kD \log (kD))$, where k is the chosen number of hashes. Each item $i \in T$ for each $T \in D$ is hashed, then for each column of data the hashes are sorted. Selecting subsets from the sorted data is efficient, requiring $O(kN)$ time, since at worst each hash is touched

once. Each transaction Id is passed to the mining phase at most once per iteration. Therefore, each item is passed at most once per iteration. We next examine the complexity of the mining phase. We will consider a single call with a partition of the data d, knowing that $D = \{d_1 \cup d_2 \cup d_3 \cdots d_N\}$ and there is no intersection among elements.

Algorithm 18 GeneratePotentialItemsetList

Input: TreeNode *Root*
Input: List L of potential Itemsets

```
 1: bool Mark = [|Root.children| == 0]
 2: for all Child c ∈ Root.children do
 3:    if c.listsize > 1 then
 4:       GeneratePotentialItemsetList(c)
 5:    else
 6:       Mark = true
 7:    end if
 8: end for
 9: if Mark then
10:    MarkNode(c,L);
11: end if
```

Algorithm 19 MarkNode

Input: TreeNode *node*
Input: List L of potential Itemsets
1: $size = node.listsize$
2: **if** !$node.marked$ and $node.depth > 1$ **then**
3: Make PotentialNode b;
4: $node.marked = true$
5: **end if**
6: **while** !$(node == NULL)$ and $node.depth > 1$ and $node.listsize == size$ **do**
7: $node.marked = true$
8: $node = node.parent$
9: **end while**
10: **if** !$(node == NULL)$ and $node.depth > 1$ **then**
11: MarkNode($node$, L)
12: **end if**

The mining phase is also $O(kD \log (kD))$. Since each node is passed once per iteration, each item is also passed only once. Let a single call pass data d. The first step builds a histogram in $O(d \log d)$ time. The histogram is then sorted. Each item from each transaction is then added to the trie at most once, requiring at worst $O($ log $d)$ time per item, since the children of a trie node are a sorted set. Generating potential itemsets is an $O(d)$ operation, since the trie can have at most d nodes, each node is visited twice, and the computation at a trie node is constant. Sorting the potential itemsets does not require more than $O(d \log d)$ time since its size is bounded by $O(d)$. Finally, each potential itemset is processed once. The processing step marks an item in a transaction, removes a item from the trie, and adds a new transaction to the data set. Each item is marked only once, each Id is removed from a list only once per item, and there cannot be more than $O(d)$ itemsets generated. Therefore, the mining phase is bounded by $O(kD \log kD)$. We use a small k, typically 8, and so we distill the overall complexity to $O(D \log D)$.

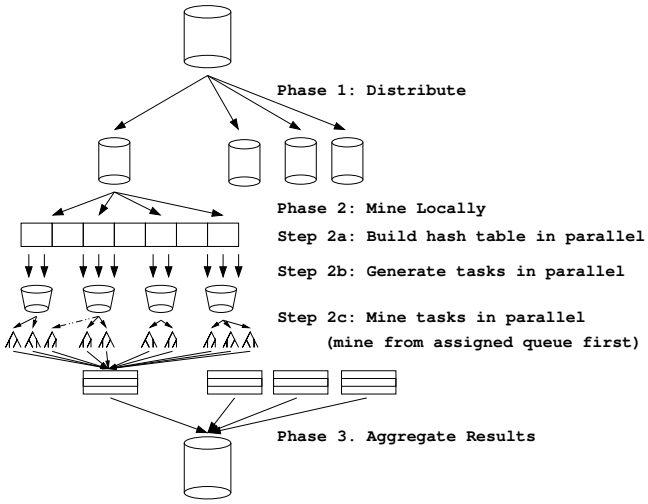


Figure 6.4: PLAM, a multi-level parallelization for LAM.

6.4.4 Multi-level Parallelization

We have designed a multi-level parallelization of the algorithm for a distributed cluster of CMP machines, named *PLAM*. It is illustrated in Figure 6.4. We assume the data resides off the cluster. The algorithm proceeds in three phases as follows. Phase one partitions the data across the cluster[18]. This can be done using our hashing mechanism described to send blocks of nodes to a machine. The machines receive an equal number of bytes. Although the mining times for each call to Algorithm 17 vary, the fact that a typical execution requires many millions of calls helps to balance this static partitioning. This is counter to typical global frequent itemset mining, where there are often only a few hundred mining tasks (frequent-1 edges). The algorithm does not require further communication, other than to aggregate the results at the end of the computation.

[18]If data already resides on the cluster then this can be omitted.

Phase two requires each machine generate itemsets locally using a CMP-parallel process. It operates using three steps – building the hash table, generating mining tasks from the table, and mining the tasks. The hash table is built in parallel by assigning each processing core a disparate and equal subset of the transactions to hash. No lock is required on the hash table as each core updates the a different set of table rows[19]. Each core then sorts its portion of the hash table. The sorted sections of the hash table are then merged using parallel merge sort. The last merge is performed by one thread, but represents a very small part of the total computation time. In step two, each core generates mining tasks in the form of *(startIndex,endIndex)* from its portion of the local hash table and inserts them into a local queue. Each local queue (one per core) has a lock. Finally, in step three these tasks are mined. Each core first empties its own queue, it steals tasks from other queues, until all queues are empty. Generally, locking presents little overhead, since a) the number of cycles required to mine a task is far greater than the number of cycles required to set the lock, and b) for a large prefix of the computation there is no contention for locks. Empirical results will corroborate in Section 6.5. The algorithm concludes by aggregating results in phase 3.

6.5 Experimental Evaluation

In this section, we evaluate the algorithm empirically. Specifically, we examine its ability to use itemsets to compress the data set, its scalability, and its overall execution time. *LAM* was implemented in C++. The code was executed on a 32

[19]The size is preallocated since the number of transactions is known.

Data Set	Vertices	Edges
UK2002	18,520,486	298,113,762
INDO2004	7,414,866	194,109,311
IT2004	41,291,594	1,150,725,436
ARABIC2005	22,744,080	639,999,458
EU2005	862,664	19,235,140
SK2005	50,636,154	1,949,412,601
UK2006	77,741,046	2,965,043,000

Table 6.4: Web graph data sets.

node cluster of Red Hat Linux machines. Each machine had two quad core Intel Xeons, 6GB RAM and one 250 GB SATA HDD.

In all experiments we use 8 hashes; more provided little compression benefit, and incurred a linear cost in execution time. The web graph data sets [12] used are presented in Table 6.4 and are publicly available[20]. The transaction data sets presented in Table 6.5 are compliments of the FIMI repository.

The empirical evaluation is divided into two sets. In the first set we compare itemset mining with *LAM* against itemset mining using closed itemsets using *Afopt* and *FPGrowth* from the FIMI repository (both return the same result set, we use the lower of the two execution times). In the second set of experiments, we delve deeper into the properties of *LAM*.

6.5.1 Comparing LAM with Closed Itemsets

In this section we compare our proposed itemset mining strategy with traditional closed frequent itemset mining. We use closed sets because closed sets performed better than both traditional itemsets and maximal itemsets when compressing data,

[20]http://law.dsi.unimi.it

Data Set	Transactions	$\|D\|$
kosarak	990,002	8,019,015
mushroom	8,124	186,852
connect	67,557	2,904,951
chess	3,196	118,252

Table 6.5: Transaction data sets.

which is our example application. Traditional frequent itemsets are too numerous to be efficient, and maximal itemsets eliminate far too many useful patterns.

Runtime Comparison

As a first comparison, we timed our *LAM* code to mine the well-known FIMI kosarak data set. The results for closed sets and *LAM* are presented in Figure 6.5. As the support is lowered, the time to mine closed itemsets grows from as little as 10 seconds at $\sigma = 3000$, to over 2000 seconds at $\sigma = 600$. If a user wishes to discover interesting rules by exploring low support values, it may be cost-prohibitive, not only because of the number of itemsets returned, but also because of the long runtimes. Our code requires 8 seconds using a single core, regardless of the support threshold, and finds large patterns with low support. We note that σ is not used in our code, and thus does not alter runtimes. When employing eight cores (shown as *PLAM8core*), the execution time is only one second.

Figure 6.6 displays the execution times for *LAM* and closed itemset mining on the EU2005 data set. For closed sets, we separate the time to mine and time to compress. For *LAM*, the times shown are for both steps. For closed itemsets, compression requires a rediscovery phase (see Section 6.3.1), which is an $O(n^2)$ operation. This second operation represented much of the cost for utilizing closed itemsets, when the

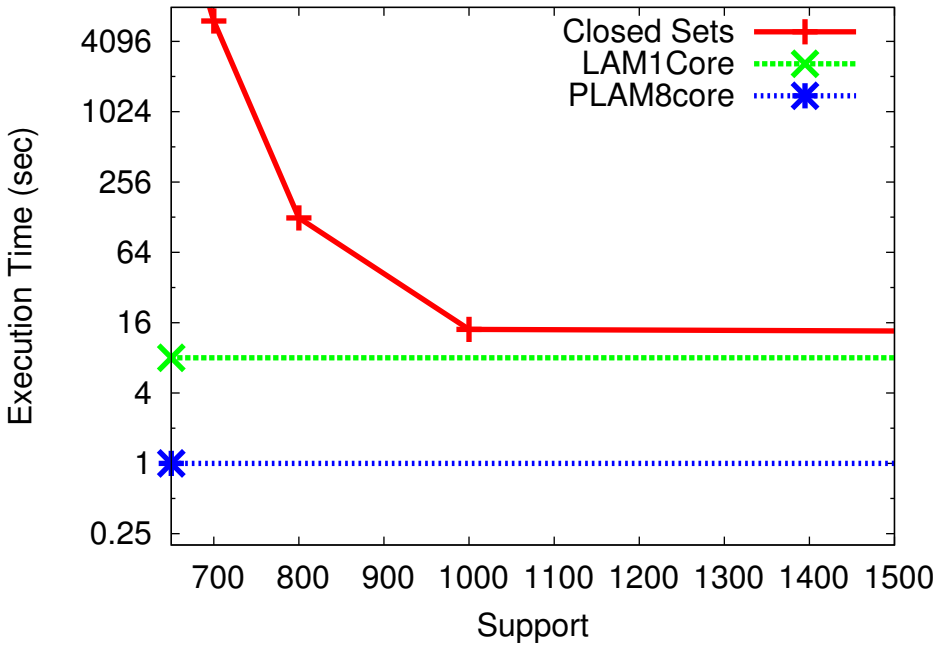


Figure 6.5: Serial execution time for closed itemsets and *LAM* vs support on the kosarak data set.

result sets grew large. Conversely, *LAM*'s localization affords inline compression (in addition to its bounded log-linear complexity), and has very low execution times. We present two *LAM* time curves, one for a single iteration and a second curve for five iterations. As outlined in Section 4, because the algorithm runs quickly, and because its uses a bias lexicographical sort, multiple iterations can be run quickly, which can produce different itemsets.

Result set Comparison

In addition, we compare the itemset results from both *LAM* and closed itemset mining. The results are presented as histograms in Figure 6.7. We mined the EU2005 data set, using multiple support levels for generating closed itemsets (larger data sets

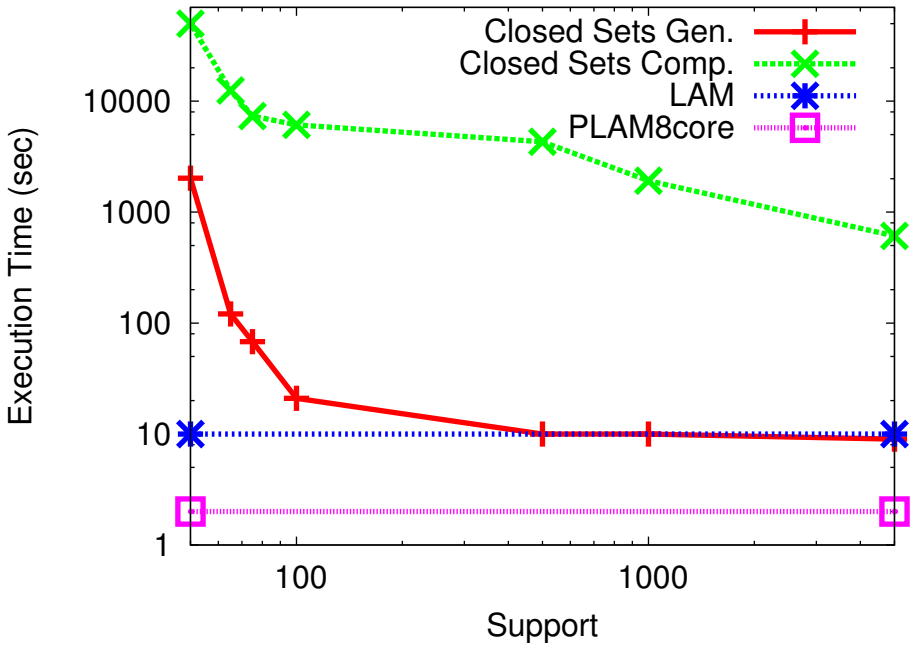


Figure 6.6: Serial execution time for closed itemsets and *LAM* vs support on the EU-2005 data set.

were cost prohibitive for closed itemsets). We also provide two result sets for *LAM*. *LAMall* is the result of outputting every pattern placed in the PotentialItemset table. *LAMTop5* is the result set from outputting only the top 10 patterns (based on length) from each record grouping. In both cases, the execution time is the same (about 10 seconds).

Several points are worth mentioning from the figure. First, it is clear that *LAM* outputs longer patterns than closed itemsets (at reasonably computable support levels). We mention that longer patterns are typically more interesting for the web graph, as they often represent link spam (particularly when the source nodes are in disparate domains). In addition, it is shown that pruning the result set with LAM is

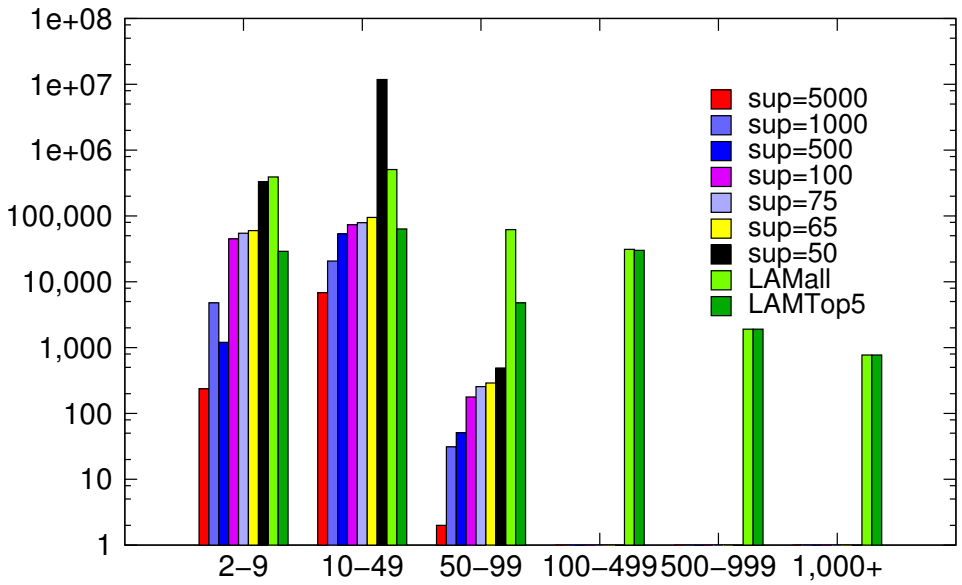


Figure 6.7: Itemset results for various supports, grouped by set size. $LAM1$ indicates one iteration.

quite simple, as the top five patterns in each group formed a much smaller pattern set. However, it maintained many of the longer, more interesting patterns.

It can be seen that a small decrease in support produces significantly more closed itemsets, from 245K at a support of 65 to 12.5M at a support of 50. Note that at a support of 50, the result set has 419M items from a data set of only 19.2M items. Further, at a support of 45, this number was increased over 10-fold before the execution abruptly halted.

6.5.2 Properties of LAM

We now examine further properties of the proposed algorithm.

Data Set	Time(min)	Itemsets Added
IT2004	27	4.5M
ARABIC2005	19	3.7M
EU2005	0.5	260K
SK2005	36	12MM
UK2006	67	18M

Table 6.6: Serial execution times (mining + compression) for *LAM* using five iterations, and the number of added transactions.

Multi-level Scalability

The *PLAM* algorithm was run on a distributed cluster of PCs to determine the strong scalability (speedup). We mine several of the web graphs from Table 6.4. The one machine execution is a single thread running on a single machine. All other executions use all eight cores on each machine, so for example 32 machines uses four machines in the cluster. We prepartitioned the data onto the machines such that each machine had about the same number of bytes. These times represent the mining times, or phase 2 of our distributed algorithm. Phase 1 depends on whether the data is prepartitioned (and is primarily limited by the properties of the channel), and Phase 3 depends on how the result set is to be used. In most cases, we use the locally generated itemsets to compress the input data, so no itemsets are actually returned. We did not include disk access time to read in the data set. This time is parallelizes when distributing across machines, but does not when scaling from one to eight cores on the same machine. As a reference, the *ARABIC2005* data set is 2.6GB and requires 35.2 seconds to load from disk on a single machine (74MB/s). Single processor execution times are presented in Table 6.6.

The scalabilities of *PLAM* for the web data sets is shown in Figure 6.8. The average scalability from 1 to 256 cores was 170-fold for the four data sets. As an example, the *ARABIC2005* data set experiences an 186-fold reduction in runtime for 256 cores (32 machines). In execution time, this is a reduction from 302 seconds to about 2 seconds, for a single iteration (not including the first disk read time). The *INDO2004* data set produced the weakest scalability numbers, up to 135-fold for 256 processors. It has a near-clique of about 7000 vertices that introduces a load balancing challenge – on 256 cores the slowest machine required more than 30% more time than the average machine. For a single machine, scaling from one to eight cores (consisting of two distinct processors) was 7.2-fold, 7.6-fold, 7.2-fold and 7.8-fold for the *EU2005*, *UK2002*, *INDO2004* and *ARABIC2005* data sets, respectively. Thus, the degradation in performance when scaling from 8 to 256 compute cores is the disparity in load balance among machines.

The compression ratios degrade slightly as the number of machines is increased. For example, from 1 to 32 machines, the compression ratio for the *EU2005* data set degraded from 4.1 to 4.0 (2.4%). This occurs because each machine mines for localized partitions within its local data only.

Compressing Other Data Sets

It is possible to use the described compression technique for any data set which can be represented as a set of transactions. We used *LAM* to compress many of the data sets from the FIMI repository. The results are presented in Table 6.7. The compression ratio is defined as the original data size divided by the new data size. *Mushroom* compressed the most at 3.57-fold. Since the algorithm captures

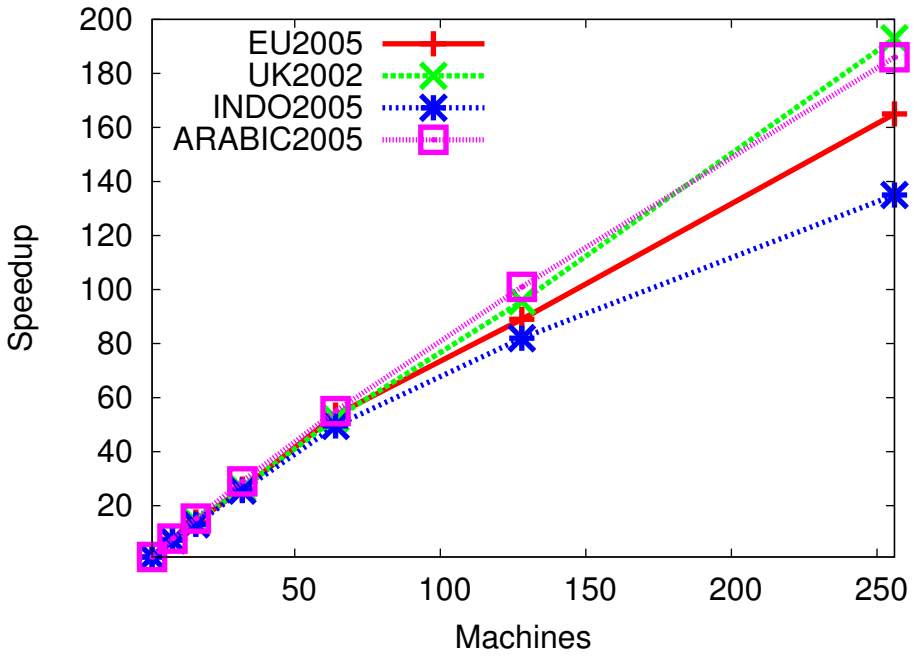


Figure 6.8: Scalability for *PLAM* on the cluster when mining large web graphs.

associativity in the input, it can be inferred that an input such as *Chess* has high inter-set associativity (2.94-fold compression) than *Kosarak* (1.28-fold). Visual inspection of the data confers this notion. A benefit of compressing with *LAM* as opposed to a global encoder such as LZ77 is that *LAM* maintains record level access.

6.6 Discussion

The number of patterns generated is of concern for truly large data sets, such as the full web graph (2TB of transaction data). By mining localized partitions of the data set, we can minimize itemset collisions, and well as discover locally high utility patterns. Also, automatically processing the itemsets as they are generated, as opposed to storing and post-processing, is essential. A benefit of our algorithm is

Data Set	Compressed Size	Ratio
kosarak	6,334,898	1.28
mushroom	52,101	3.57
connect	1,122,846	2.58
chess	40,109	2.94

Table 6.7: Compression statistics for other common itemset data bases.

that the compression can be easily embedded into the mining phase for two reasons, both afforded by the localization. First, transaction Id lists tend to be shorter and require little state to maintain, and second because the number of patterns discovered per localized area is smaller. The former may be mitigated in closed mining using a vertical format, but TIDlist structures are often stored on disk, which will be slower than the in-memory solution provided by *LAM*. Even if this cost was zero, *LAM* affords better compression in less time.

Alternate Solutions We investigated more traditional partitioning algorithms for the front end of our framework, such as *kMeans*. This did not perform well. To be scalable, it requires a low number of centers, which did not offer the necessary granularity to discover small, localized patterns in the graph.

Future Directions It is our belief that there are use cases beyond graph compression for itemsets where only a lower bound is given for the global support of a pattern. First, such sets could be used to generate document signatures for certain phrases in a corpus (e.g. proper nouns). We note that in general the barrier to entry for most large data information retrieval challenges is often an algorithm which scales $O(N \log N)$ with increasing input dimensions. Second, one could simply return the most interesting or a set of interesting patterns, suitably defined from each localized

partition. Our proposed algorithm can generate a rather long pattern for a niche subset of the data, which could then be used as a group signature, requiring very little execution time. Comparisons with extant approaches to generate profiles would be worth evaluating in this regard. Third, the local partitions generated by LAM and the subsequent itemsets discovered could be effectively leveraged for classification based on itemset patterns. This could have the potential advantage over extant algorithms in terms of its ability to classify within sub-spaces, i.e., local partitions. These are all avenues of future research that we would like to pursue.

6.7 Conclusion

We have described a parameter-free algorithm for discovering interesting patterns in very large data sets. By relaxing the requirement for exact global counts, we can bound the computational complexity for discovering interesting itemsets to $O(D \log D)$. The solution executes in two stages, both of which are efficient. It discovers patterns in localized portions of the data which are not typically found by traditional mining strategies. We also develop a scalable multi-level parallelization of the algorithm for clusters of CMPs. Finally, we provide empirical results which illustrate the utility of the technique. In the following chapter, we will use our proposed mining process to compress large scale web graphs.

CHAPTER 7

A PATTERN MINING APPROACH TO WEB GRAPH COMPRESSION USING COMMUNITIES

7.1 Introduction

Conservative estimates place the size of the web to be 11.5 billion pages[21], and more than 300 billion links. More aggressive estimates suggest a number closer to 30 billion pages[22]. The scale is so large that many engineering techniques simply are not effective. Significant effort has been expelled to develop information retrieval algorithms which scale to these proportions, tackling challenges such as deep page indexing, rapid user response times, link graph construction and querying, and efficient storage. The work we present here targets the latter two issues.

The connectivity of the web has been shown to contain semantic correlation amongst web entities [39]. For example, if many pages with similar hyperlink text point to the same page, it can be inferred that page contains information pertinent to the text. Algorithms such as HITS [67] and PageRank [15] formalized this notion. Gibson, Kleinberg and Raghavan [41] first suggested that the web was composed of fairly organized social structures. Naturally, the search community has developed

[21] http://www.cs.uiowa.edu/ãsignori/web-size/

[22] http://www.pandia.com/sew/383-web-size.html

dedicated systems to allow algorithm designers to query the web's structure, called *Connectivity* or *Link Servers*. Connected components in the structure, such as dense bipartite graphs or cliques can be inspected for interesting correlations. Search engine optimizers (SEOs) are known to exploit this fact by injecting additional, less-useful links to improve relevance rankings, a practice commonly referred to as *link spam*. Efficient community discovery via connectivity servers has been shown to improve detection of such spam [42].

We can generally classify link queries as either streaming or random access queries. Streaming queries touch most or every node in the graph, while random access queries often have unpredictable access patterns. For example, *PageRank* is typically implemented in a streaming, data-push fashion. However, many interesting questions are more efficiently answered by random access algorithms. Example queries include *How many hops from page X is page Y*?, *Is page Y a member of a close-knit community of pages*?, and *Do we think X could be link spam*? If the graph does not fit in main memory, frequent random seeks to disk can render the system useless, since hard disk access is five orders of magnitude slower than access to RAM. Therefore, significant research has focused on compressing the link structure of the web.

Existing strategies leverage the fact that if the vertices in the graph are ordered by their corresponding URLs, then from node to node there is significant overlap in outlinks. In addition, it is common for the outlinks (in sorted order) to have small differences in destination Id (due to the ordering). This can be exploited by encoding the difference in successive Ids instead of the Id itself, a process called *gap coding* [90]. However, global ordering of URLs is fairly expensive, both because URLs average 80 bytes, and because the web graph is updated with new URLs by crawlers at least

daily. The compression technique we illustrate does not depend on any ordering or labeling of the source data.

The goal of knowledge discovery and data mining (KDD) is to extract useful information from data sources, often in the form of recurring patterns. While not completely ignored, frequent pattern mining has not been significantly leveraged to address web search challenges in the literature. The most likely cause is that pattern enumeration is quite expensive in terms of computation time. In this work, we leverage pattern mining to both compress the web graph, as well as provide kernel patterns for web community discovery. We make use of the mining algorithm described in the previous chapter to mine for patterns in web graphs efficiently. We will illustrate the efficacy of the imposed complexity constraint, as the mining phase is executed several million times when compressing a single graph.

It is our observation that rather than gap coding the graph, we can use community structure to compress the graph. It has the two-fold benefit of both providing excellent compression ratios, as well as affording constant-time access to the discovered communities for a given URL. To accomplish this, we cast the outlinks (or inlinks) of each node as a transaction, or itemset, and mine for frequent subsets. We find recurring patterns, and generate a new node in the graph called a *Virtual Node*. The virtual node has the outlinks of the target pattern. We then remove the links from the nodes where the pattern was found, and add the virtual node as an outlink. In many cases, we can represent thousands of edges with a single link to a virtual node. In addition, the pattern is a bipartite clique, which often has semantic meaning. The reason is that it is common for two competing companies to not link to each other, but third parties will often link to both, thus detecting inter-domain patterns [68].

This phenomena may not be captured by clique discovery. Also, bipartite graphs within a domain often depict a web template pattern, such as navigation bars. For large, multi-domain hosts whose URLs do not share a common prefix, these template patterns can help to classify the page. Also, since the method adopts a global view of the data, the discovered communities are often inter-domain groups. Finally, many community discovery algorithms can use virtual nodes as seeds to discover larger communities.

Specifically, the contribution of this work is a link server compression and community discovery mechanism with the following properties.

- The compression ratio is high, often higher than the best known compression techniques in the literature.
- Community discovery queries can be performed quickly, often in $O(1)$ time.
- The compression supports random access to the web graph without decompressing.
- It does not require sorted and adjacently labeled vertices, such as by URL, and thus supports incremental updates easily.
- It is highly scalable, capable of compressing a 3 billion edge graph in 2.5 hours on a single machine.
- It owns time and space complexities of $O(E \log E)$.
- It is flexible, clearly divided into two phases; algorithm designers are free to incorporate alternate algorithms easily.

7.2 Background

We present relevant background information for the interested reader. Related research has been presented in Chapter 2.

7.2.1 Connectivity Servers

There are several fundamental challenges to address when designing a link server. Two of the primary challenges are the size of the graph and the ability to support random walk queries.

In the graph representation, each page is represented as a vertex in the graph; hyperlinks are directed edges. Without compression, each link requires log(—V—) space, or 36 bits, thus the graph requires about 2 terabytes. For computations which access every edge, such as static rank, the graph can be stored on disk with a reasonable penalty to computation costs. For search queries, seeking on disk is cost-prohibitive, so it is desirable to store the 2TB in RAM. Most commodity servers house 4 to 16GB of RAM, suggesting a need of 128 to 500 machines. Compression can reduce the number of machines significantly.

The outlinks (and possibly the inlinks) of each vertex are stored serially in RAM. An additional structure is kept which stores V offsets, one for each vertex's first link. We point out that this structure requires approximately $O(V \log E)$ bits, or if compressed, $O(V \log \text{comp}(E))$ bits, where *comp(E)* is the total cost for all the compressed links. Each halving of the bits per link value only reduces the cost of the offset structure by a small margin (for example, 1/32th). We will add to the cost of this index, because we add additional vertices to the graph. However, on average we add only about 20% more vertices, and does not change the size significantly (see

Section 7.4.2). The majority of the web graph compression literature report bits-per-link values. We will do the same, to provide for comparisons. One can see that as we compress the edges, the roughly constant size of the offset structure becomes the bottleneck to lowering the total storage costs.

In addition, if reverse indexing of the actual URLs is desired, this will incur a fixed cost as well. Typically, the URL table is queried only at the conclusion of the computation, and only if page-level details are desired. Finally, it may be desirable to end all indexable data on a byte boundary, which will incur padding costs [11].

7.3 Virtual Node Miner

In this section, we detail our algorithm, called *Virtual Node Miner, or VNMiner*. It is built upon *LAM*. The premise of our method is that the overlap in link structure present in the web graph can be summarized by adding a relatively small number of virtual nodes. These virtual nodes are a one-level indirection of intersecting bipartite graphs. As an example, consider the two graphs in Figure 7.1. The top of the figure displays a dense bipartite graph, where the source nodes (labeled S) all share a subset of destination links. By introducing an additional node in the graph (VN), 19 of the 30 shared links can be removed.

Algorithm 20 VNMiner

Input: A graph G
Output: A compressed graph G'

```
1: for i=0 to NumberOfPasses do
2:    C = Cluster nodes in G
3:    for all c ∈ C do
4:       Patterns P = Find patterns in c
5:       for all p ∈ P do
6:          Virtual Node v = elements of p
7:          Add v to G
8:          for all node ∈ c do
9:             if p ⊂ node.links then
10:               node.links = node.links − p + v
11:            end if
12:         end for
13:      end for
14:   end for
15: end for
16: Encode each node in G
```

We cast the problem of finding common links for nodes as a frequent itemset mining challenge, where each node's outlinks (or inlinks) is a transaction. The number of labels is then the number of nodes in the graph. The desired minimum support is then two, which presents a significant challenge. However, we do not require the full enumeration of frequent patterns, nor for the exact counts of each pattern. As discussed earlier, exact itemset mining is an exponential computation. Using a simple intersection of the provided links is too conservative, as many subpatterns are unnecessarily disregarded. To circumvent the nearly impossible task of mining hundreds of millions of data points at once, we leverage the approximate itemset miner described in Chapter 6 to discover high utility patterns. We then remove the pattern, generate a virtual node for it, and reiterate. We can also use the same process to compress the added virtual nodes.

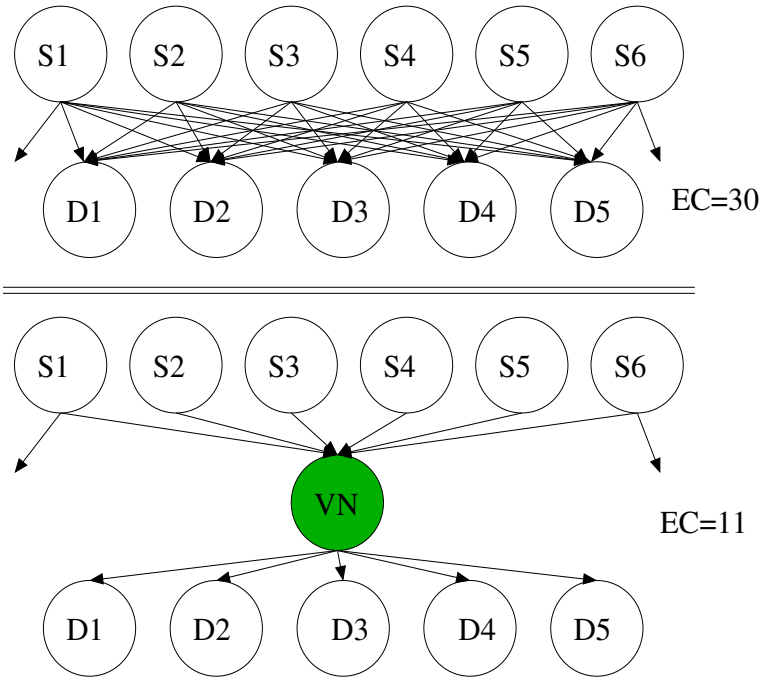


Figure 7.1: A bipartite graph compressed to a virtual node, removing 19 of 30 shared edges.

Pseudo code outlining *Virtual Node Miner* is provided as Algorithm 20. Lines 1 and 15 iterate based on user input. Each iteration is an attempt to compress each vertex in the graph. Line 2 forms clusters of the vertices. Lines 3-14 loop through each cluster and compress them. Line 4 mines for patterns in the current cluster, c. This step uses the approximate mining procedure discussed in Chapter 6. Then, for each pattern found, lines 5-13 attempt to use the pattern to compress the cluster's vertices. Line 6 generates a virtual node. Line 7 adds the virtual node to the graph. Lines 8-12 compress the vertex outlink lists associated with the current pattern. Line 10 removes the outlinks associated with the current pattern from each node that contains the pattern, and adds a single outlink to the virtual node. We name the

clustering process in line 2 the *Clustering Phase* and the mining process in lines 4 - 12 the *Mining Phase*.

7.3.1 Complexity Analysis

The complexity of the proposed approach is of particular interest, due to the size of the problem at hand. In each iteration, we use our approximate itemset miner from Chapter 6. For a graph $G = (V, E)$, the clustering phase is $O(kE \log E)$, where k is the chosen number of hashes.

Therefore, the overall algorithm is $O(E \log E)$.

7.4 Empirical Evaluation

We evaluate our algorithm empirically to gain an understanding of its performance. *Virtual Node Miner* is characterized according to the compression capabilities, the quality and quantity of the generated virtual nodes, and the time to run the process. The implementation is C++ using STL on a Windows PC running Windows Server 2003. The machine is a 2.6 GHz Dual Core AMD Opteron(tm) with 16GB of RAM. For this evaluation, only one core is used, as the implementation is currently serial. In all trials we use 8 hashes, as more did not improve results. Also, the trials provided are for outlinks; we have investigated inlinks and observerd slightly better compression values.

The data sets used are provided in Table 7.1. In large part, these are the same web graphs introduced in Chapter 6.

Data Set	Nodes	Edges	Edges/Node
WEBBASE2001	118,142,155	1,019,903,190	8.63
UK2002	18,520,487	298,113,762	16.09
IT2004	41,291,594	1,150,725,436	27.86
ARABIC2005	22,744,080	639,999,458	28.13
EU2005	862,664	19,235,140	22.29
SK2005	50,636,154	1,949,412,601	38.49
UK2005	39,459,925	936,364,282	23.72
UK2006	77,741,046	2,965,197,340	38.14

Table 7.1: Web data sets.

7.4.1 Compression Evaluation

We evaluate the compression capabilities of the algorithm. In all cases we include the cost of the virtual nodes and their inlinks when reporting compression values. First we use only virtual nodes to remove edges from the graph. Figure 7.2 (left) illustrates the results for each of the eight data sets. The vertical axis is the ratio of total edges in the original graph divided by the edges remaining after compression (including the virtual node edges). Virtual nodes perform best on recent data sets which have a higher edge to node ratio. For example, the *UK2006* data set has more than 7 out of every 8 edges removed from the original graph for a compression of 7.1-fold. The compression decays with each pass, as fewer patterns are found. In all cases ten iterations (or passes) saturates compression.

We compared the lexicographically sorted clustering scheme with super-shingles [17] and found that the former bested the latter by 15% on average. Super-shingles are well suited for finding dense subgraphs. However, in our experiments, we found that their requirements for a match are too stringent for compression purposes. The break even point for compression to leverage similarity between two nodes is roughly

two shared edges between them. These matches that are useful for compression have a much larger distance when compared with those produced using super-shingling.

In Figure 7.2 (right) we compress the remaining edges with Δ coding followed by Huffman coding. From the figure we can see that overall compression is quite high, up to 14.3-fold on the *UK2006* data set, and higher on others. In all cases, the maximum compression occurs within four iterations of the algorithm. This occurs because removing edges degrades the efficiency of gap coding. In Figure 7.3 this is illustrated for the *UK2006* and *SK2005* data sets. While virtual node compression continues to improve for several passes, gap coding techniques use more bits per edge. We implemented both Δ Huffman coding as well as ζ coding. An advantage of ζ codes is that they are flat (no need to search a tree for the code). While Huffman coding is optimal, it requires longer decoding times and also must store the decoding tree, which we truncate for efficiency. In these trials, the table was truncated.

A summary comparing *Virtual Node Miner* (labeled *VNM*) with the *WebGraph* framework is provided in Table 7.2. The proposed technique is quite competitive (we do not have maximum compression values for the *UK2006* data set at this time). Since these values are in bits/edge, they do not account for the cost of the offsets. We note that as the number of bits per edge decreases, offset overhead becomes significant. For example, the *IT2004* data set averages about 28 edges per node. If those edges are compressed using only 2 bits each (Table 7.2), then the representation requires 56 bits per node for the edges, and 26 bits per node for the offset into the edge array. Finally, the URLs are also stored, which at a 10-fold compression requires approximately 80 bits per node. Thus, link servers supporting random access and reverse URL lookup

Data Set	WebGraph	VNM	# VN
WEBBASE2001	3.07	3.01	13.3M
UK2002	2.22	1.95	4.11M
IT2004	1.99	1.67	4.48M
ARABIC2005	1.99	1.81	3.62M
EU2005	4.37	2.90	265K
SK2005	2.86	2.46	11.2M
UK2005	1.70	1.42	7.1M
UK2006	NA	1.95	17.1M

Table 7.2: Bits per edge values for *WebGraph* and *Virtual Node Miner (VNM)*, and the total number of virtual nodes generated (#VN).

will find that *Virtual Node Miner* and *WebGraph* are comparable *from a compression standpoint.*

7.4.2 Virtual Node Evaluation

The number of virtual nodes generated to support maximum compression is reported in Table 7.2. On average, the number of virtual nodes added to the graph is about 20% of the original number of nodes, which does not introduce a significant additional overhead into the offset array. The *UK2006* graph required 22% of virtual nodes to reach maximum compression. These additional nodes add to the offset cost slightly, but reduce the number of total edges, as shown in the low bits/edge values. *WEBBASE2001* has a low edge to node ratio (8.63). As a result, does not compress well, and only generates 13.3M virtual nodes (11%).

Figure 7.4 provides insight into the size and distribution of the virtual nodes generated. The left figure displays the mean number of virtual nodes referenced by an original node. After one pass, on average each original node has 0.7 outlinks to virtual nodes. After ten passes, each original node has 1.45 outlinks to virtual nodes.

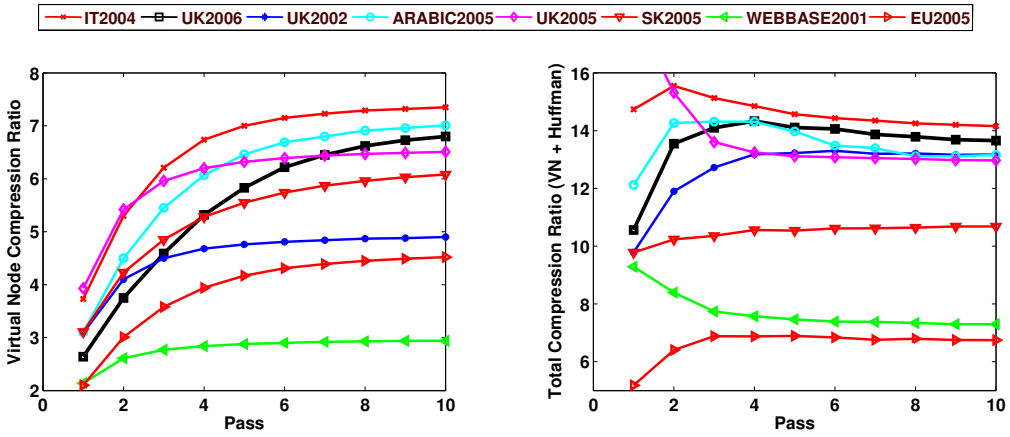


Figure 7.2: Compression afforded by virtual nodes (left), and total compression (right).

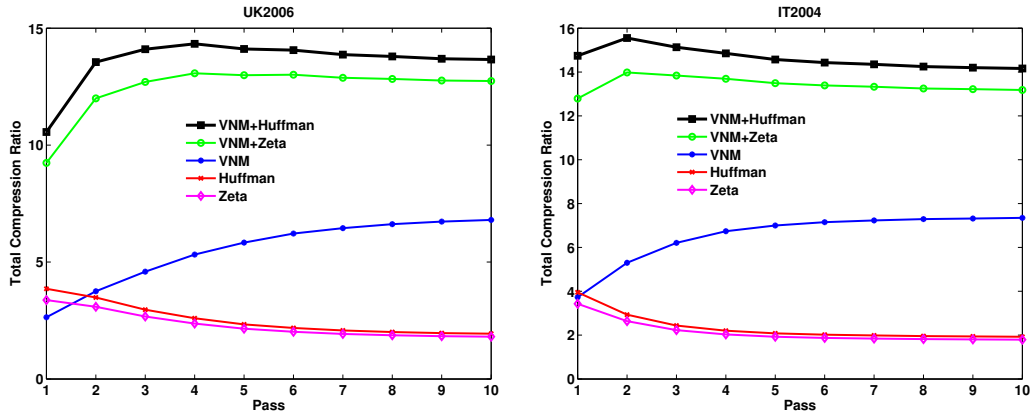


Figure 7.3: Compression factors for the *UK2006* data set (left), and the *IT2004* data set (right).

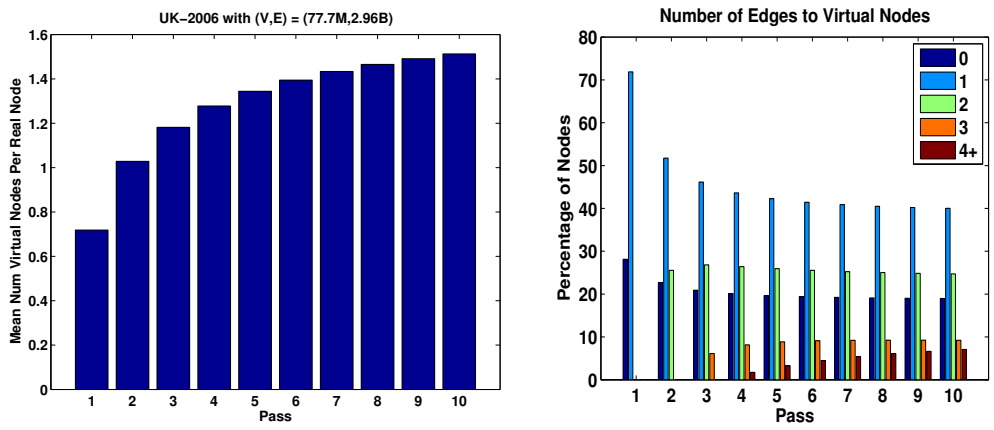


Figure 7.4: The number of virtual nodes per iteration (left) and the cumulative average number of virtual nodes (right).

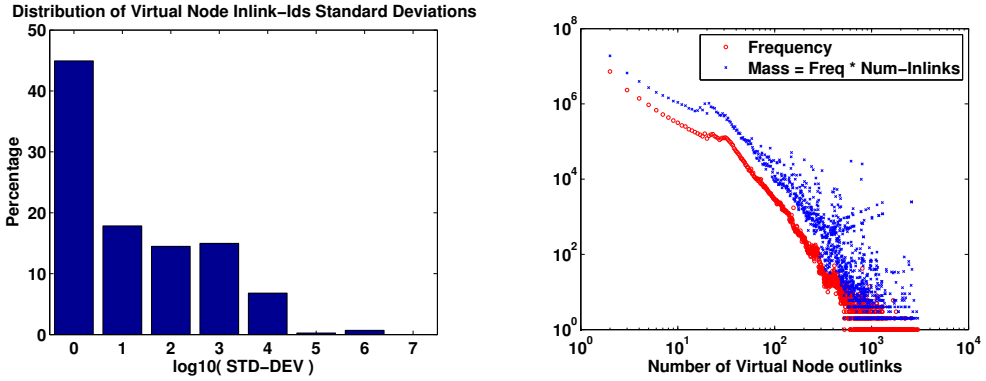


Figure 7.5: The standard deviation of the ranges for inlinks of virtual nodes (left), and size/mass values for the virtual nodes (right) for the *UK2006* data set.

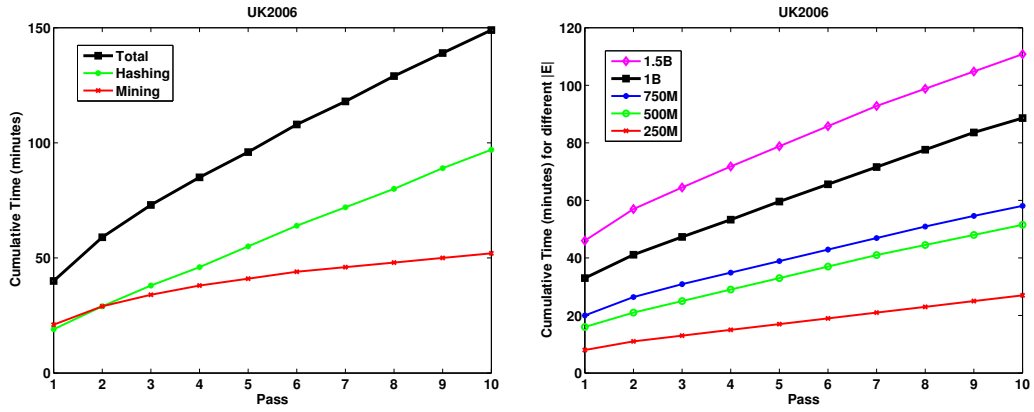


Figure 7.6: Execution time for each phase (left), and execution time as a function of graph size(right) for processing the *UK2006* graph.

Thus, even after ten passes, the average number of dereferences to virtual node lists is only 1.45, requiring approximately 600 nanoseconds. To compare, *Webgraph*'s maximum compression requires 31,000 nanoseconds. The right figure displays the distribution of the number of virtual nodes for each original node as a function of the number of passes of the algorithm. The maximum number of virtual nodes for any given original node cannot exceed the number of passes. It can be seen that the worst case number of dereferences is a small percentage; at ten passes less than 7% have more than four virtual nodes in their outlink lists.

In Figure 7.5 (left), the standard deviation of the ranges of the inlinks of the virtual nodes is plotted. This plot provides a means to evaluate what percentage of the patterns captured by *Virtual Node Miner* would be captured by windowing schemes. Most windowing schemes use a window size less than ten [90]. At any window size less than ten, more than 50% of the patterns have ranges which would not be found. Figure 7.5 (right) plots the distribution of the mass of the virtual

nodes. Mass is the number of inlinks using the pattern multiplied by the number of patterns with the associated length. The x-axis represents the length of the pattern.

We also illustrate the benefits of longer patterns found by *LAM* in this next experiment. We calculate the compression ratio as we continuously make use of longer patterns. Again, compression ratio is the defined as the number of items in the original data set divided by the number of edges in the compressed data set. Figure 7.7 displays the effects on the larger UK-2006 data set. The X-axis is the pattern length and the Y-axis is the *cumulative* compression ratio. It can be seen that patterns of length 20-60 have the greatest benefit, comprising almost 50% of the compression. This is evident as the slope of the curve is highest in this region. It is evident that longer patterns do in fact provide significant benefit. As shown, patterns longer than 500 provide an additional 10% compression. Patterns of lengths greater than 1000 provide a modest improvement, although final compression values are less impacted by patterns in this region due to their lower frequency.

7.4.3 Execution Time Evaluation

Experimental evaluation clearly demonstrates that the algorithm is scalable. As shown in Figure 7.6 (left), the execution time for the *UK2006* graph (which has about 3B edges) requires less than 2.5 hours. Note that the times in the vertical axis in both figures are cumulative. In this trial, 89 million separate function calls were made to the mining phase. All other data sets in this study required less computation time. For data sets which exceed main memory, the algorithm is run in batches. We have found that *Virtual Node Miner* can be run effectively on machines with 2GB of RAM. If desired, the number of passes could also be trimmed, since the compression

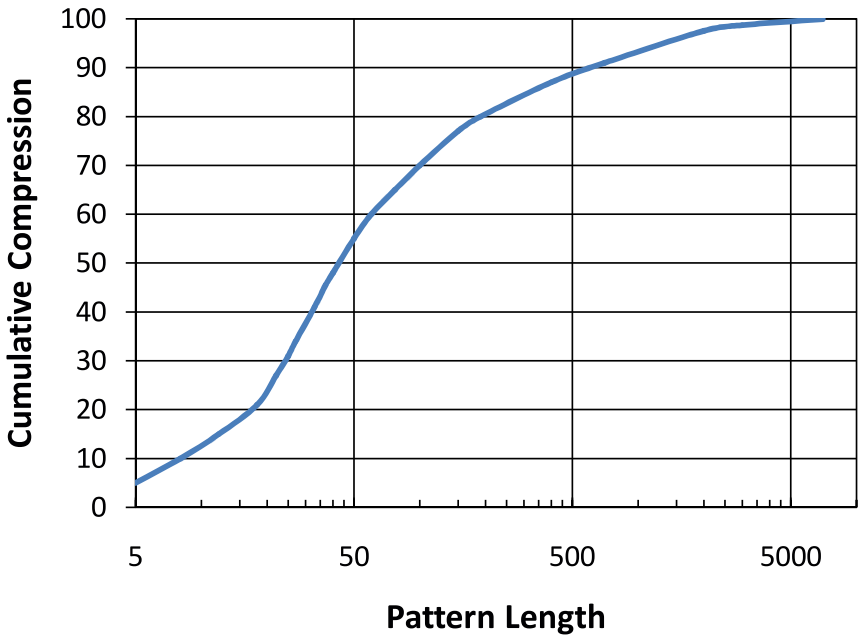


Figure 7.7: Compression afforded by pattern length (cumulative).

degrades with each pass. In Figure 7.6 right, the graph is partitioned into blocks of 125 million edges. *Virtual Node Miner* scales linearly as the size of the graph increases, from 27 minutes on 250M edges to 110 minutes on 1.5B edges. The first pass requires more time than the others primarily because the data set is read into memory. For example, Pass 1 required 47 minutes for 1.5B edges, whereas pass 2 only required 11 minutes.

7.4.4 Community Seed Semantic Evaluation

Finally, we examine several of the typical community seed patterns discovered represented by virtual nodes. For this experiment, we output the members of communities whose vertex Ids flagged several simple heuristics when compressing the *UK2006* data set. We measured the size of the pattern, the number of vertices linking into the pattern, the standard deviation of the Ids for those inlinking vertices, among other properties. Since the public web graph data is sorted by URL, virtual nodes with high standard deviations in vertex Ids represent communities whose members do not reside in the same host domain. We then generated histograms by bucketing Ids into 100 bins and visually inspected the graphs. Figure 7.8 depicts four example communities. Table 7.3 lists the starting vertex Id, ending vertex Id, and a sample URL for each cluster in each community. Each community is separated by vertex Id ranges, where applicable. The size of each community is then the sum of the sizes of its ranges. For example, community 16 consisted of 10,182 web pages.

The first image, community 11, consists of just one range of contiguous vertices, from 11,393,132 to 11,394,190 totaling 1,058 pages. It was flagged by our pattern filter because it is a rather large (over 1,000 outlinks) pattern and had many inlinks

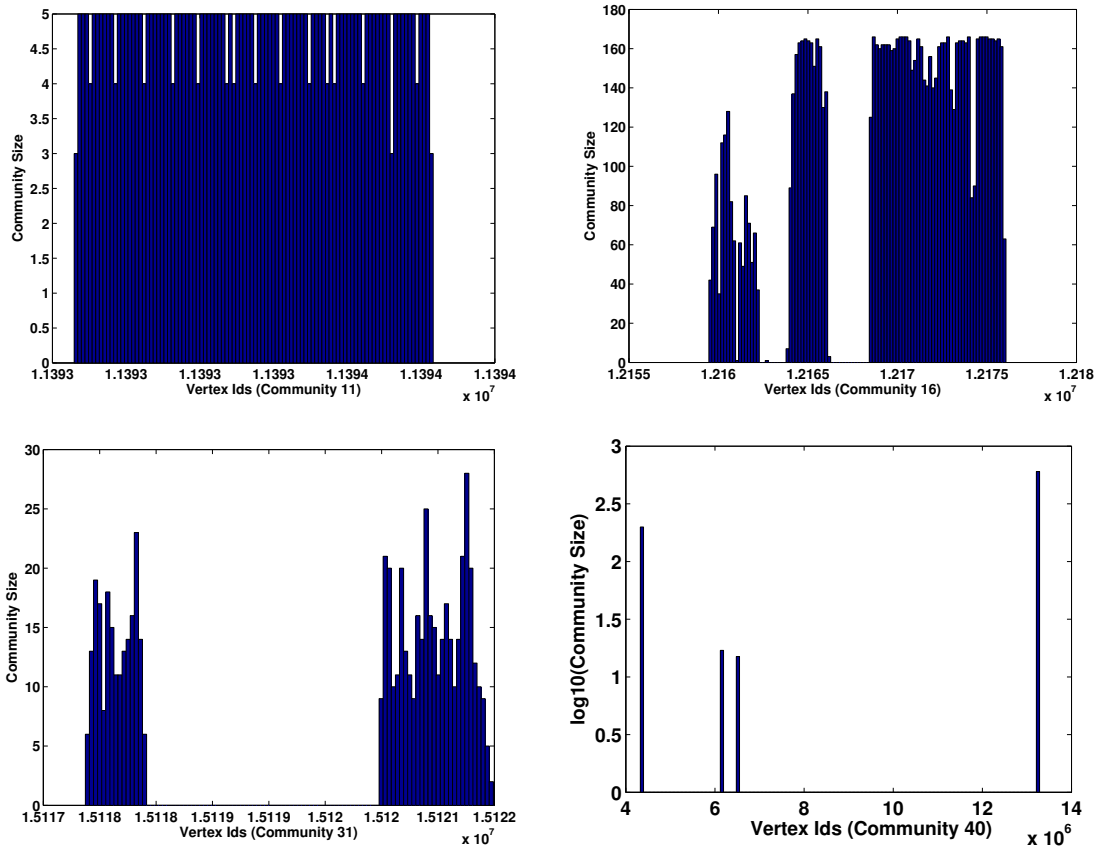


Figure 7.8: A sample of community seeds discovered during the compression process (from top left to bottom right, community 11, 16, 31 and 40).

(again, over 1,000). Upon inspection it appears to be a link farm for "loan69". The pages linking into the farm direct PageRank towards advertising pages showing simple search results.

The second image is community 16, consisting of three distinct groups. It appears to be an online cellular equipment sales company using multiple domains – *mobilefun.co.uk* and *mobilepark.co.uk* – using common web templating. It was flagged because the three groups all have significant size and the total range is large. We found this pattern common as well. In fact, they also use *mobileguru.co.uk* which happened to sort closely to *mobilefun*.

The third image is community 31. This pattern is also from *mobilefun.co.uk* but includes a prefix *ringtones*. The site is a search engine for downloadable cellular content. The site appears to use a common trick to increase crawler activity. Dynamic web pages are artificially simulated in server scripts to appear to be static pages within a directory structure. For example, the local page */family-guy-Mp3-Ringtones.htm* is actually the results of a query to the search input form, which when sent to the server uses the querystring */search.php?term=family+guy+mp3&s=0*. Crawlers often downgrade querystring terms in URLs, which would have resulted in fewer instances of the *search.php* page in the index.

The fourth image is community 40. It consists of four distinct ranges. The number of vertices in each range is [651 273 4,662 11,463] respectively. This pattern is clearly the result of a sorting artifact, where the four ranges are actually four sub domains linking to common files, such as header and navigation links. These domains are listed in Table 7.3. We found this template pattern to be quite common. The total distance between the vertices in the pattern is about 9 million in a graph of about

Seed	Start Id	End Id	Size	URL
11	11,393,132	11,394,190	1,058	http://loans69.co.uk/ site-map/index_c_426.html
16	12,159,546	12,162,237	1,164	http://mobilefun.co.uk/products/6789.htm
16	12,163,934	12,166,105	1,957	http://mobilefun.co.uk/sale/Memory.htm
16	12,168,466	12,175,979	7,061	http://mobileguru.co.uk/ /Mobile_Technology_resource.html
31	15,117,891	15,118,397	204	http://ringtones.mobilefun.co.uk/ family-guy-Mp3-Ringtones.htm
31	15,120,491	15,154,304	474	http://www.mobilefun.co.uk/ sale/Samsung-S500i.htm?sortby=nameasc
40	4,368,219	4,368,870	651	http://comment.independent.co.uk
40	6,167,376	6,167,649	273	http://education.independent.co.uk
40	6,506,773	6,511,435	4,662	http://enjoyment.independent.co.uk
40	13,236,350	13,247,813	11,463	http://news.independent.co.uk

Table 7.3: Properties of the ranges from the community seeds shown in Figure 7.8.

77 million vertices (note that both axes are log scale). This type of pattern could not be easily found using reference windows, as most practical solutions of this form limit the window size to less than ten vertices. It has been suggested to sort domains in reverse order, to close the gap in vertex Ids for sub-domains. For communities 31 and 40 this looks to be effective, however for communities similar to 16 this would aid the process of discovery.

7.5 Discussion

Virtual Node Miner has been shown to be efficient, compressing web graphs many-fold in a short amount of time. Also, the fraction of the graph held in main memory is configurable, so it can support a variety of machine configurations. The algorithm executes in two distinct phases, both of which are amenable to parallelization. More than half of the execution time is spent computing hashes, which are data level

independent. The sorting at the end of the first phase is a reduce operation and needs synchronization. The second phase mines nodes in groups. These mining function calls are independent. Each call has a disjoint set of graph nodes that can be processed in parallel. Virtual nodes found by parallel mining operations are concatenated at the end of the second phase of each pass. Although the cost for each mining call will vary with the number of edges passed, in iteration of compression there are over 2 million such calls, and we expect a simple work queue to load balance well.

When examining compression ratios and execution times, we can conclude that the proposed technique is competitive with the state of the art, namely the Webgraph Framework[11]. We now discuss several advantages.

Virtual node mining based compression does not require a specific coding mechanism for edges. Competing coding schemes can be applied, such as ζ codes [10] or Huffman codes. The input data need not be sorted by URL. In fact, the proposed algorithm works equally with either sorted or unsorted data. This affords incremental graph updates, which occur at least daily for most web crawl systems. In contrast, gap-based coding schemes using reference windows are highly sensitive to the URL ordering [10, 90].

The virtual node based compression scheme natively supports popular random walk based computations such as PageRank [15], spectral graph partitioning [5], clustering [29, 101], and community finding algorithms [106]. These algorithms can be implemented on the compressed graph without requiring decompression. We briefly present a description of PageRank on the compressed graph. In compressed PageRank, each iteration is split into a sequence of sub-iterations. The push operations in each sub-iteration are of two types: push operations towards virtual nodes and

push operations towards original (non-virtual) nodes. In the first sub-iteration, original nodes push probabilities to both original and virtual nodes. In subsequent sub-iterations only virtual nodes push partial sums (accumulated over previous sub-iterations) to original and downstream virtual nodes. The number of sub-iterations is equal to the longest sequence of virtual nodes and never exceeds the number of passes used during graph compression. From Figure 7.4 (left) we note that each node on average has less than 1.45 virtual nodes. Thus, the total number of push operations after the first one or two sub-iterations is very small. Note that the recursive references to virtual nodes produce a directed acyclic graph (DAG) of virtual nodes. For streaming efficiency, one can envision storing the virtual nodes in a topologically sorted order.

It is also worth noting that the proposed compression scheme not only supports computations in compressed form, but also has potential for speeding up computations. It is easy to see that if the number of edges undergoes a five-fold reduction during compression, then the total number of push operations also undergoes a five-fold reduction. The five-fold reduction in push operations can be expected to produce up to a five-fold speedup. The compression scheme stores tightly connected communities as shallow hierarchies of virtual nodes. Using this representation one can efficiently answer community queries such as: *Does page X belong to a community*? *Who are the other members in the community page X belongs to*? *Do pages X and Y belong to the same community*? etc. While answering these queries we can expect to drop nepotistic/intra-domain links [36].

We consider the mining process presented in this work to be approximate because it produces support values for itemsets which may be lower than the actual support

in the data set. Also, the same itemset may be found multiple times. We do not attempt to reduce these collisions – instead we compress the virtual nodes as well. The penalty is then one additional outlink, and one additional dereference.

Finally, we believe that the approach is generally applicable, and easily implementable. Alternate clustering methods can be easily inserted, as well as alternate pattern mining kernels. For example, the algorithm designer is free to use more than $O(E \log E)$ time in the mining process if tolerable.

7.6 Conclusion

This work proposes a scalable approximate data mining mechanism to compress the web graph for link servers, which incorporates the formation of global patterns. We feel these patterns can be used as seeds in the community discovery process. The algorithm exhibits high compression ratios, and scales to large data sets. The end representation averages only slightly more than one dereference per vertex for random walks. It also has several properties not present in existing compression technologies: i) it requires no particular ordering or labeling of the input data, ii) it can find global patterns in the input data, and iii) it supports any of the several available coding schemes. In addition, this work introduces a process for mining itemsets in log-linear time. Directions for future work include a parallel formulation, and a mechanism to grow larger communities from the discovered seed patterns.

CHAPTER 8

A PLACEMENT SERVICE FOR DATA MINING CENTERS

8.1 Introduction

Advances in data collection technology have led to the creation of large scale data stores. Examples abound ranging from astronomical observational data to social interaction data, from clinical and medical imaging data to protein-protein interaction data, and from the multi-modal content on the World Wide Web to business and financial data routinely collected by enterprises. A fundamental challenge when working with very large data is the need to process, manage and eventually analyze it *efficiently*. A commodity cluster of nodes with a low-latency, high-bandwidth interconnect and with high-capacity commodity disks offers a cost-effective solution to this challenge. However, leveraging the features of such clusters or data centers to deliver end-to-end application performance is still fraught with difficulties.

Three of the most significant issues when employing distributed clusters for data mining are i) localizing computation (maximizing independent computation and minimizing communication), ii) generating a balanced workload, and iii) the need to be fault tolerant (data loss, hardware failures). The placement of data onto a tightly

coupled system area network can have a significant impact on the performance of the application in all three of the challenges above, and is therefore the target of this article.

There are several orthogonal dimensions to consider when developing data placement policies. For example, a placement policy might attempt to place closely related pieces of data together to facilitate localized computation. Along another dimension, a policy might favor redundant data placement to facilitate fault tolerant computing. This replication may also be used to improve computation runtimes. A third dimension is disk balance. Support for flexible placement policies is thus highly desirable. In this article we describe the design, development and evaluation of such a flexible placement service for next generation data analysis centers. While the placement service is fairly general-purpose, in this article we place emphasis graph and network structured datasets. Such datasets are fairly ubiquitous, and processing and analyzing them is particularly challenging. For example, computing *betweenness* on the web graph, which is conservatively estimated to be on the order of 20 billion nodes and 500 billion edges, requires a combination of distributed computation, distributed placement, and clever compression techniques.

The central features of our proposed placement framework include

- low-complexity policies to improve similarity between records on a machine,
- efficient, indexable data redundancy
- a flexible hashing mechanism for placement which allows user-specified similarity metrics

- a simple-to-use programming interface which supports a variety of data input types

We evaluate our placement framework on several large, real datasets drawn from the web domain. We show the service maintains the majority of the nearest neigbors for a random sample of records when partitioning data across a cluster of 1024 PCs. This holds for both document data as well as graph data, maintaining up to 65% more neighbors than typical distribution strategies. Also, it reduces the cuts in a distributed graph from 99% using a round robin scheme to only 7% on 1024 machines. For pattern mining, the local partitions contain over 10-fold more closed patterns at the same support levels. Finally, we illustrate the effectiveness of the placements schemes when compressing the data, reducing the data footprint over 14-fold while still maintaining indices into each record.

8.2 Preliminaries

In this section we describe the target environment for our service, as well as the existing approaches.

8.2.1 Model

We develop this placement service for a distributed cluster of PCs executing a variety of data mining applications. The input data is assumed to be in a flat file (two or three files, as described later) which can be accessed by at least one node in the cluster. The challenge is to distribute the data to local disks in the cluster such that a) the load is relatively balanced (from a bytes per node perspective), while b) applications making use of the data have lower runtimes than simple uniform

random placement provides. In addition, we would like to incorporate robustness in the placement so that if one node in the cluster is unavailable, that the data can be rediscovered with minimal processing time, automatically in the best case. We assume the existence of a low-level API to address communication between nodes. In our implementation, this interface is MPI.

8.3 Placement Algorithms

We now describe the details of the proposed placement algorithms. Generally the goal is to group similar nodes together, where similarity is a function of the intersection between records. However, as noted below, the hashing mechanism is designed to allow any user-specified measure.

8.3.1 Hashed-based Placement

Hashed-based placement uses a hash function to produce a signature for each record in the data set. The signature is typically a fixed width of size K[23]. Users may provide a hashing function, or use one of the several described below. We hash each of the M records of the input data K times to produce an $M * K$ matrix. We then sort M lexicographically. This sort is typically quite fast since M is designed to fit in RAM[24]. Next we traverse the matrix column-wise, grouping rows with the same value. When the total number of rows drops below a user-provided threshold, or we reach the end of the hash matrix, we box the record Ids as a partition of the data, and assign it to a machine in the cluster. For example, suppose in the first column there is a contiguous block of 200,000 rows with the same hash value. We then compare

[23]In practice we use values between 8-32, noting that larger values limit subsequent bias.

[24]If it doesn't, we make multiple Ms that do, performing the partitioning process on each M

the second column hashes of these 200,000 rows. For each distinct hash value in the second column, we inspect the number of rows with that value. If cardinality is below a user-defined threshold, we pass it to a *partition assigner*; otherwise we inspect the third column hash values for the selected rows, and so on. The lexicographic sort biases the sampling left-wise in the matrix. Therefore, using a sort has the negative effect of placing unwanted weight on the first hash–rows which do not match in their first hash but in all others will not be placed in the same group–but sorting reduces the complexity from an $O(n^2)$ operation to an $O(n \log n)$ operation. For large data sets, this effectively changes a computation from intractable to tractable.

Pseudo code for hash-based placement is provided as Algorithm 21. Here we assume the input is a data set of sets – we describe data representations in Section 8.6.1. Lines 2-9 build the $M * K$ matrix, with the outer loop executing once for each record in the data set. The inner loop shown in lines 4-7 hashes each dimension of the current record and adds it to a vector v. Line 8 then adds v to the matrix M for the current record. The matrix is then sorted lexicographically in line 10. Lines 15-21 generate lists of consecutive indices in the matrix with the same hash value. Lines 16 and 17 increment the index for the current hash value. These indices are then processed by Algorithm 22. at line 19. This process is a slight abstraction from the min-wise independent permutations presented in Chapter 6.

Algorithm 21 Hash

Input: Data set D
Input: Hash family $h()$
Input: Number of hashes K
Output: A partition $P = \{p_1, p_2 \dots p_N\}$

```
 1: Matrix M
 2: for all rec ∈ D do
 3:    vector v
 4:    for all k ∈ K do
 5:       Hash h = h(rec, k)
 6:       v_k = h
 7:    end for
 8:    M_rec = v
 9: end for
10: Sort M Lexicographically
11: start = 0
12: end = 0
13: col = 0;
14: currentHash = HashTable[end][col]
15: while end ≤ |D| do
16:    while currentHash == HashTable[end][col] do
17:       end++
18:    end while
19:    GetList(start, end, col + 1)
20:    start = end
21: end while
```

In Algorithm 22, line 1 checks the size of the partition. If the list is large, lines 5-10 scan the next column in the matrix for equal hash values. For each distinct hash value, it generates a recursive call (line 10). This call uses the next hash column by incrementing the index. However, if the index range is below a threshold, or the process has reached the last column in the matrix (line 1), a call to a partitioning algorithm is made (line 2). Several assignment algorithms are available, depending on the data type (see Section 8.6.1. We describe mechanisms for graphs and flat records provided as Algorithms 23 and 24.

Algorithm 22 GetList

Input: StartIndex *st*
Input: EndIndex *end*
Input: Index *index*

```
1: if end − start <THRESHOLD or index = K − 1 then
2:    AssignPartition(st,end)
3:    return
4: end if
5: currentHash = HashTable[st][index]
6: while st < end do
7:    while currentHash==HashTable[st][index] do
8:       st++
9:    end while
10:   GetList(st, end, index + 1)
11: end while
```

The *AssignPartitionForGraphs* procedure provided in Algorithm 23 chooses an appropriate machine for a set of records. In lines 2-13 a histogram is built of the locations of the items of each record passed. Note that these items in the record correspond to outlinks in the graph. Thus it is desired to locate the machine with the greatest number of outinks, and assign the passed records to that machine. Line 4 dereferences from the hash table index to a record in the data set. Lines 5-9 loop through each outlink id of the current vertex, t. Line 6 finds the current assignment for the current outlink id. If it has been assigned to a machine (line 7), the location for the vertex is added to the histogram. Full machines cannot contribute to the histogram. List P retains the record Ids, and the *loc* array stores the assignment of each record in the data set. Lines 15-21 send the records to the machine with the most weight in the histogram and update the weights of that machine. Line 16 sends the vertex to the assigned machine. Line 17 updates the statistics (the machine

assignment) for that vertex. Line 18 increases the current load for the machine where the vertex was sent.

The *AssignPartitionForTrans* algorithm distributes small groups of items based on a histogram of the elements on each machine. Although seemingly similar to the previous algorithm, we do not dereference items in the records to discover their location, because the items are not assumed to be references to other records. Instead, we maintain a running multi-set of each machine's current records. We use a heuristic extension of a well-known algorithm in statistics, which selects any element from a stream of elements with equal probability by selecting the first element and then for every subsequent element, replaces the existing element with probability $1/N$, where N is the number of elements seen thus far. We generate our multi-set by maintaining a buffer of up to W elements. When the buffer reaches size W, we remove elements with $1/2$ probability. Then, the next record inserted onto the machine updates adds its elements to the multi-set with $1/2$ probability. Subsequent halving of the multi-set lower the probability of a record adding its elements to the multi-set by $P/2$. This set is then used as a signature for the machine. A set of records' distance to the machine is defined as the intersection between the union of the items in the records and the machine's signature set.

Algorithm 23 AssignPartitionForGraphs

Input: startIndex st
Input: endIndex end
1: List P
2: Histogram h
3: **while** $st < end$ **do**
4: RecordId t =*HashTable[st].recId*
5: **for each** $e \in t$ **do**
6: Machine $m = loc[e]$
7: **if** $m > -1$ and *weight[m]*$<MAX$ **then**
8: $h = h \cup e$
9: **end if**
10: **end for**
11: $p = p \cup t$
12: $st++$
13: **end while**
14: Machine $m = Max(h)$
15: **for each** $t \in p$ **do**
16: Send $Tran[t]$ to m
17: $loc[t] = m$
18: $weight[m] += |t|$
19: **end for**

Pseudo code is provided as Algorithm 24. Lines 1-3 initialize local variables. Lines 4-10 build a set S of elements of the records to be assigned. Line 5 dereferences from the hash table index to a record in the data set. Lines 7-9 loop through each id of the current record, t, and add them to a set S. Lines 13-19 find the machine with the largest intersection with S. Sig is an array of the multi-set signatures. Line 15 compares the proximity of the current set to the highest current intersection, and assigns to it if the intersection is greater (lines 16 and 17). Lines 20-24 send the records to the proper machine and update the proper signature by calling *UpdateSignature*. Line 21 sends the record to the appropriate machine m. Line 22 updates the loading factor for m. Line 23 updates the multi-set signature for m.

Pseudo code for *UpdateSignature* is provided as Algorithm 25. In line 1, this algorithm first calculates the possibility of adding elements to the multi-set by using $Prob[m]$, which essentially stores the number of times the signature has been divided. Lines 2-9 perform the (approximate) window halving, if the size of the signature has exceeded the maximum size. Lines 3-7 loop through the elements of the signature and with 50% probability, retain the element. Line 8 halves the probability of including a new item. Lines 10-12 then add the elements of the new record.

Min Hashing

The framework for hash-based distribution allows for any user-defined hash function, such as Cosine, Jackknife, etc. We provide two predefined functions. The first is min-hashing[16], uses probabilistic sampling on the data set. Min-wise K-way hashing is a process by which a pseudo random element is chosen from a set with consistency. One uses K independent permutation functions to select k elements from the set. Each permutation function places a different order on the elements in the set, and the first element is considered the signature of that set for that permutation. Performing k permutations arrives at a signature of length k. It has been shown elsewhere [16] that min-hashing asymptotically approaches the Jaccard coefficient with increasing k. The Jaccard coefficient is given below.

$$j = \frac{A \cap B}{A \cup B} \tag{8.1}$$

Thus if two elements have many similar hash values, then the probability that they have many items in common is high. For many data mining applications, such as graph clustering, etc, this similarity measure is desired.

Algorithm 24 AssignPartitionForTrans

Input: StartIndex *st*
Input: EndIndex *end*
Input: Histogram *centers*[]

```
1: List P
2: Histogram h
3: Set S
4: while st < end do
5:    RecId t = HashTable[st].recId
6:    P = P ∪ t
7:    for each e ∈ t do
8:       S = S ∪ e
9:    end for
10: end while
11: maxid = −1
12: max = −1
13: for each m ∈MS s.t. weight[m]<MAX do
14:    prox=Intersection(S, Sig[m])
15:    if prox > max then
16:       max = prox
17:       maxid = m
18:    end if
19: end for
20: for each t ∪ P do
21:    Send Tran[t] to m
22:    weight[m]+ = |t|
23:    UpdateSignature(t,m)
24: end for
```

Approximate Hashing

Approximate hashing provides a mechanism to partition data such that a prefix tree can be constructed in multiple parts independently [20]. The idea is to sample the data to create a histogram, then on a subsequent full pass use a geometric binning procedure to move each transaction into a particular bin. All the items in bin b_i sort before items in b_{i+1}. If the data is highly skewed, which can be obtained from the

initial scan, then each item in the sorted histogram receives twice the storage as the following bin.

Since transactions typically contain many items, the second item in a transaction partitions the sub-bins as it partitioned the original bin space. For example, suppose items a and b are the two most frequent items in the data set. If bin $b_1 \ldots b_{10}$ were assigned to item a and $b_1 1 \ldots b_1 5$ were assigned to item b, then bins $b_1 \ldots b_5$ would contain transactions having both *a and b*. We generalize the procedure to fit our hashing model above. For approximate sorting, the *kth* hash function selects the *kth* most popular item for the hashed transaction. The algorithm runs in linear time in the size of the data set and log-linear in the size of the sample.

Instead, we leverage domain knowledge and the frequency information collected in the first scan to approximately sort the frequent transactions into a partition of blocks. Each block is implemented as a separate file on disk. The algorithm guarantees that each transaction in block_i sorts before all transactions in block_{i+1}, and the maximum size of a block is no larger than a preset threshold. By blocking the frequent data set, we can build the tree on disk in fixed memory chunks. A block as well as the portion of the tree being updated by the block will fit in main memory during tree construction, reducing page faults considerably.

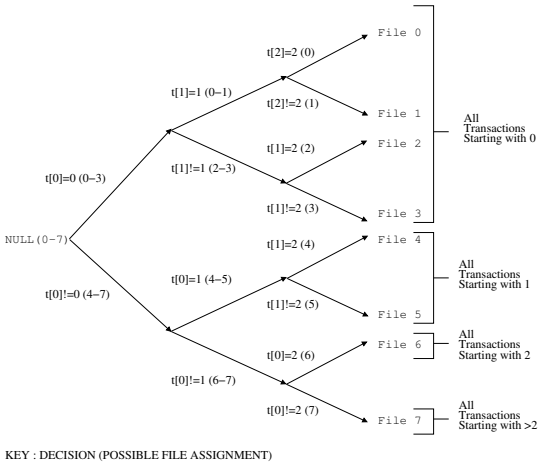


Figure 8.1: Geometric Partitioning Decision Diagram, for 8 bins.

Algorithm 25 UpdateSignature

Input: Machine m
Input: RecId t

```
1: if rand()%Prob[m] == 0 then
2:    if |Sig[m]| > W then
3:       for each w ∈ Sig[m] do
4:          if rand()%2 == 0 then
5:             Sig[m] = Sig[m] \ w
6:          end if
7:       end for
8:       Prob[m]* = 2
9:    end if
10:   for each e ∈ t do
11:      Sig[m] = Sig[m] ∪ e
12:   end for
13: end if
```

8.3.2 Other Placement Routines

Sorted Placement: The interface provides for an arbitrary sorting routine to be used. The mechanism is to then sort the data per the passed function, and partition

the data evenly based on the total number of bytes assigned to each machine (subject to record boundaries).

Round Robin Placement: It may be the case that the user desires a relatively uniform distribution of data on each machine. The round robin scheme is advantageous in this case, because unlike randomly choosing a machine for each cluster, the pattern is regular. Therefore, no index file is needed, and each node can calculate at runtime the machine mapping for every record.

8.4 Data Redundancy

We address fault tolerance by adding a replicated but compressed form of each partition. The default placement of the compressed partition is the next machine Id in the mapped cluster. For example, define the partitioned data as

Finally, user-defined placement can be implemented by passing a custom sorting function pointer.

$$P = \{p_1, p_2, p_3 \ldots, p_N\} \tag{8.2}$$

where i is the machine id for subset p_i. Then the compressed data is defined as

$$C = \{c_1 = f(p_N), c_2 = f(p_1), \ldots c_N = f(p_{N-1})\} \tag{8.3}$$

where $f(p)$ represents the compressed form of p.

Two compression schemes are provided. The first uses approximate frequent itemsets in records to locate redundancy, and then compresses these itemsets into a pointer to the set. By maintaining only one copy of the itemset, all but one occurrence of the pattern can be stored using one 4 byte value (if there are less than 4 billion records). The algorithm is efficient when compared to traditional itemset mining – it runs in

$O(E \log E)$. In addition, it affords compression ratios up to 15-fold for many large data sets, as shown in Chapter 7. Two caveats are that i) the mechanism is dependent on correlation of items to be effective, and ii) the order of elements within a record is not respected. This latter issue acceptable for adjacency lists and transactional data but will offend input data where order implies information, such as bitwise feature vectors. In addition to affording improved compression rates over other methods, it also allows direct indexing into the records in compressed form, since all compression patterns exist within the boundaries of a record. Thus, for space conscious users, this format can serve as both the main data file as well as the backup file.

The second compression scheme is to compress the partitions using traditional techniques such as *LZ77* [125]. The placement location of the compressed chunk is the same as the previous method. *LZ77* often provides excellent compression ratios for a variety of data. The main drawback is that the compressed file must be fully uncompressed to be used.

8.5 Remote I/O

In many cases, data residing in the RAM of another machine is faster to access than local data on disk. This can be particularly useful when i) the local data set is too large to reside in local RAM, and ii) when local intermediate data (data generated during computation) exceeds local RAM. For this reason, we offer remote I/O functions to move data to and the RAM of a remote machine. The details of the API are presented in Section 8.6.4.

8.6 Placement Service API

In this section we present the service interface to the programmer. A high-level overview is provided as Algorithm 26.

8.6.1 Data Representation

The user provides input data as transactions (or records) using two or three files. The first input file is a binary string of the elements in the records. The second input file is an index which stores offsets for each record. For example, the *ith* value in the offset file is the starting address of the ith record in the record file. The length of the *ith* record is simply $index[i+1] - index[i]$. The last record uses the length of the record file as the secondary boundary. The optional third file is an XML meta data file. Each record in this third file is of the form *<Rec>text</rec>*. Child nodes of *<Rec>* may be added, as they simply pass through our representation unaltered. We now describe how the first two files are used to represent the many input formats of typical data mining stores.

One large graph or tree can leverage the above input format via an adjacency list, where each index is a consecutive node in the graph. Urls and other meta data are stored in the meta file. A set of graphs uses the index file to position each graph in the record file. The record file first lists V values representing node labels, then a '-1', followed by E tuples in the form of *SourceNode,DestinationNode,EdgeLabel*. A set of trees follows the same format. If an index file only has one value, then it is assumed that each record has length of that value.

Currently 6 data types are available. These include 1 byte, 2 byte, 4 byte, 8 byte, 4 byte float and 8 byte floats. We require this size for the elements in a record for hashing purposes. Again, any ASCII data is stored in the meta file and is not hashed.

8.6.2 Bootstrapping

The service is started with by calling $Ss :: StartService()$. The save path and input data filename must first be provided. Currently, it is assumed that the index file can be obtained by appending '_index' and the meta data file can be obtained by appending '_meta'. Each registered data input is maintained in the user's *Manifest* file. The manifest file is an XML file containing the pertinent parameters necessary to verify the full data set is present. It includes the filename, the save path used, the number and names of the machines used for the original partitioning, the distribution type, the data type, the local data sizes and the index type.

Algorithm 26 Placement Service Programmer Interface

```
1: bool StartService(int rank,int nTasks);
2: bool ShutDown();
3: void PrintManifest();
4: void ResetManifest();
5: bool SetFilename(std::string filename);
6: bool SetSavePath(std::string filename);
7: bool AddData(int indexMode, int distMode,int dataSize, int dataType);
8: bool AddData(int indexMode, int distMode, int dataSize, int dataType, void*
   Comparator);
9: bool ReplicateData(bool UseHashCompression);
10: bool DeleteData();
11: bool DataExistsInAnyManifest(std::string filename);
12: bool RetrieveData(File* data, File* index);
13: bool RetrieveData(File* data, File* index, File* metaData);
14: bool VerifyData();
15: bool VerifyDataFilesExist();
16: bool VerifyNumberOfServers();
17: int AddRecordRemotely(int destMachine, int recordNumber, int recordLength,
    int* record);
18: int RetrieveRecordRemotely(int destMachineHint, int recordNumber, int*
    record);
19: int RetrieveRecordRemotely(int recordNumber, int* record);
20: enum {Transactions,Graphs,Trees,OneGraph,OneTree};
21: enum {Range,LocalItems,GlobalKnowledge};
22: enum {SingleMachine, Random, RoundRobin, MinHash, Hashed,ApproxHash,
    Sorted, Euclidean, Serial};
23: enum {1byte,2byte,4byte,8byte,4byteF,8byteF};
```

8.6.3 Adding Data

Users add data by calling one of the two *AddData()* functions on lines 7 and 8. They provide a distribution mode (line 22), an index mode (line 21), a data size (line 23), a data type (line 20), and possibly a function pointer.

Distribution Modes: There are eight distribution modes, which are specified during the function call using the enumeration on line 22. *Single Machine* stores all the records on a single machine. *Random* places records randomly on the cluster. It

requires either the *Local* or *Global* index types, as ranges are not practical. *Round robin* places record r on machine r modulo M. It does not require an index. *Min hashing* uses the distribution algorithm outlined in the previous section, as does *Approximate sorting*. *Hashed* allows for a custom hash function family. Data added in this manner must use the function on line 1, and pass the hashing function family as function pointer which accepts a record id and K and returns a pointer to the hashes. *Serial* partitions the data evenly across the cluster starting with the first $|D|/M$ records being placed on machine 0. *Sorted* does not hash, but sorts the data directly, based on the comparator passed. This comparator may access meta data from the meta data file by accessing the field *service.meta[i]*. *Euclidean* uses standard kMeans to distribute the data set.

Index Modes: The index mode determines the amount of information stored at each machine regarding the location of the records. Four index modes are available. *Ranged* provides the ranges of records for each machine. All machines have knowledge of the location of every record. This mode is most efficient for the *Serial* distribution mode. *Local items* provides in the mapping file each local item id, which maps in order to the index file. The user must query the other machines for knowledge of records off the local machine. Lastly, in the *Global knowledge* index mode, each machine has the location of each file in the data set.

8.6.4 Remote I/O

Several functions serve to leverage the distributed RAM in the cluster. *AddRecordRemotely()* allows a machine to insert a record into the RAM of another machine. The motivation is that distant RAM has lower latency than local virtual memory

(disk-based memory). The function takes four arguments: the machine Id, the record number, the record length, and a pointer to the record at the local machine. It may be that the calls fails for the chosen machine. Therefore, the return value is the Id of the machine where the data was finally placed. Also, the user may pass a machine Id of -1, in which case the data will be placed at a random machine.

RetrieveRecordRemotely() is used to retrieve a record from a remote machine. This record may be either from a previous call to *AddRecordRemotely()*, or merely from a *RetrieveData()* call. Three parameters are passed: the target machine, the record number, and a pointer to store the data locally. Alternatively, a user may simply pass the second two parameters, and the service will query to locate the data. In practice, these functions have float signature counterparts.

Algorithm 27 Example Program Adding Data

```
 1: Mpi_Init(&args, &argv);
 2: Mpi_Comm_size(Mpi_Comm_World, &nTasks);
 3: Mpi_Comm_rank(Mpi_Com_World, &myRank);
 4: std::string inputFile(argv[1]);
 5: Ss service;
 6: service.SetFilename(inputFile);
 7: service.SetSavePath("/scratch/buehrer/");
 8: service.StartService(myRank,nTasks);
 9: service.AddData(Ss::MinHash, Ss::GlobalKnowledge);
10: service.PrintManifest();
11: if myRank==0 then
12:    cout << 'Size=' << nTasks << endl;
13: end if
14: File *data, File *index;
15: bool res = service.RetrieveData(data,index);
16: UInt64 size = GetFileSize(index);
17: for int i=0;i<size;i++ do
18:    vector<int> *rec = GetRec(data,index,i);
19:    Process_Rec(rec);
20: end for
21: service.ShutDown();
22: Mpi_Finalize();
23: return 0;
```

8.6.5 A Typical Use Case

We provide a simple example to illustrate the utility of the proposed service API, as shown in Algorithm 27. Note that there is currently no install procedure – the user simply includes the C++ header files for the service. Lines 1-3 are typical to MPI programs. In line 6 the programmer provides a pointer to the input data. Line 7 sets the save path for partitioned data. This path must be accessible by each node in the cluster. The example uses local scratch space. Line 8 starts the service. This call opens the local manifest file on each machine, which stores the relevant information for all data sets which have been assigned to each node. Line 9 adds the data set to

the cluster. The user chooses a distribution enum and an index enum, in this case using min-wise hashing and a complete index on each machine. Lines 11-13 merely print MPI information. Line 14 intializes file pointers. Line 15 retrieves the local data pointers, using the *Retrieve()* function. Once handles to the local files are retrieved, processing records is the responsibility of the user. As an example, the size of the index file can be used to determine the number of records in the local data file (line 16). Lines 17-20 process each record. Service state is maintained on a per-user basis. Each data object is stored in an XML file in the user's local scratch space on the cluster [25]. Subsequent calls to process the data use the *Retrieve()* function without adding the data source.

8.7 Evaluation

In this section, we evaluate the algorithms empirically. All algorithms were implemented on Linux in C++. The cluster used is the same as that from Chapter 3. It contains 64 machines connected via Infiniband, 48 of which are available to us. Each compute machine has two 64-bit AMD Opteron 250 single core processors, 2 RAID-0 250GB SATA hard drives, 8GB of RAM, and runs the Linux Operating System. In all experiments we use 8 hashes; more provided little compression benefit, and incurred a linear cost in execution time. The data sets used are presented in Table 8.1. The web graphs were discussed in Chapter 6. *NLM* is a 2006 snapshot of the National Library of Medicine's MEDLINE-PubMed database (English documents only) of abstracts. There are about 8.5M documents included here, normally distributed in length. Each record is a document, and each item is a word id (stop words have been removed).

[25] We do not address security issues at this time

Data Set	Records	Items	Item/Rec	size	Type
UK2002	18,520,487	298,113,762	16.09	1.2GB	web graph
ARABIC2005	22,744,080	639,999,458	28.13	2.4GB	web graph
EU2005	862,664	19,235,140	22.29	77MB	web graph
NLM abstracts	8,509,325	588,802,208	69.2	2.2GB	documents
Webdocs	1,692,000	397,000,000	234	1.48GB	documents

Table 8.1: Data sets.

The *Webdocs* data set was obtained from the *FIMI* Repository[43]. It is a collection of web documents, using word Ids as the items in each record. Most general-purpose placement middlewares will use either i) user-directed placement or ii) a random or round robin placement. Therefore, in this section we will make use of the round robin placement scheme for comparison purposes.

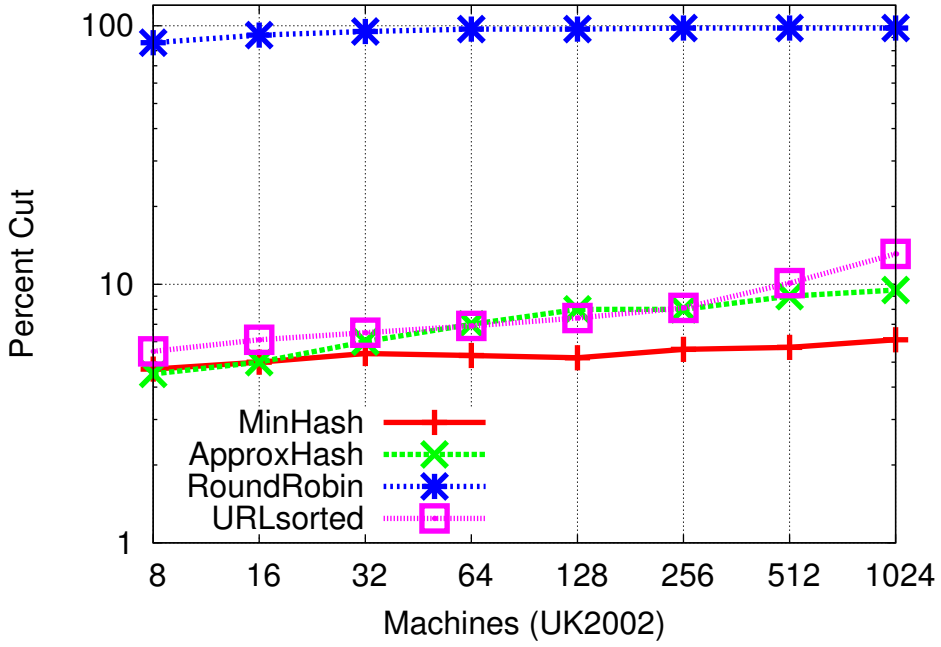
8.7.1 Partitioning Large Graphs

We analyze the ability of our hashing to partition a large web graph data on a cluster. The quality metric used is the percentage of graph edges that reside on the local machine divided by the total number of edges in the graph. So a *percent cut* of 25 means that 3 out of every 4 links point to vertices residing on the local machine. We use two graphs, namely *EU2005* and *UK2002*. Figure 8.3 depicts the results of partitioning *EU2005*, and Figure 8.2 presents the results for *UK2002*. Min-hashing is compared to round robin as well as a custom sorting function, which is to sort the graph by the URL. We include URL ordering because it is a popular and effective custom sorting routine for grouping similar nodes in web graphs. For *EU2005*, We also compare to *Metis*[65], a well-known graph partitioning algorithm by Karypis and Kumar (the *UK2002* data set required too much memory). As can be seen, results

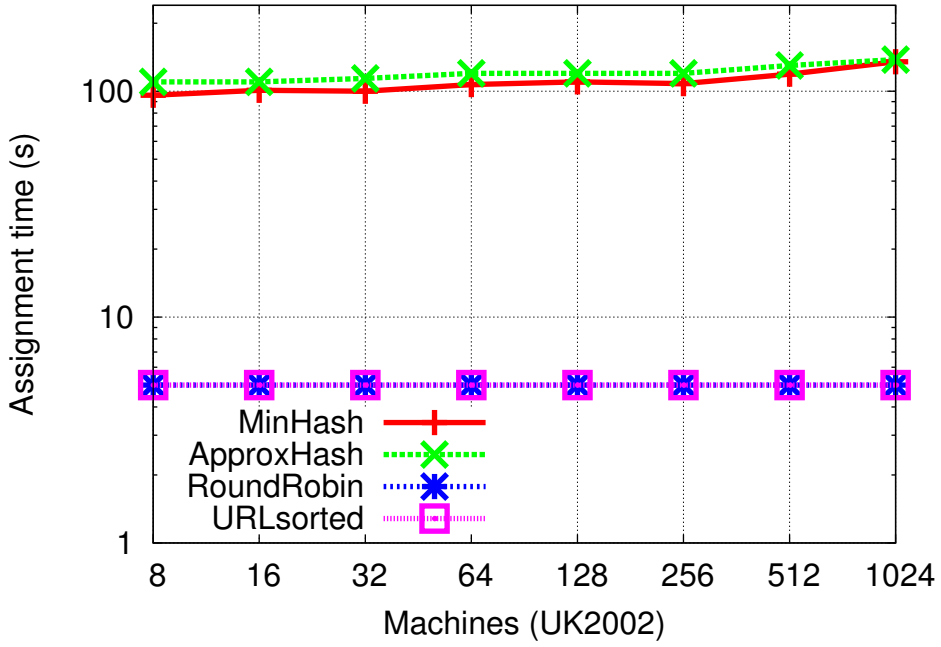
afforded by our general-purpose hashing are quite competetive. For example, hashing incurs less than half the edge cuts for *EU2005* on 1024 machines (32% vs. 68%) when compared to URL ordering, and is also better than *Metis* (47%).

Graph partitioning under hashing does suffer load imbalance. We inspected this challenge by using four different balance factors and measuring the load distributions. Roughly speaking, the balance factor is the probability that we assign the the records passed to Algorithm 23 to the location with the highest outlink correlation. The graphs in Figures 8.4 and 8.5 provide the standard deviation (and the average) of the load for a given balance factor, as well as the percent edge cut. It can be seen that the standard deviation can be controlled, but at the cost of increased edge cuts. There is clearly a tradeoff between the balance of the load and the improvement in placement. Still, in all cases the total number of cut edges is vastly lower than what is offered by a round robin placement scheme. Another point to consider is that 1024 machines is a large number of machines to use to partition the smallish (77MB) *EU2005* graph. We note that *kMetis* had a balance factor of 8.27 on 1024 machines, meaning the largest machine had 8.27-fold more data than the average.

The offline costs associated with each partitioning mechanism are presented in Figures 8.3 and 8.3 (bottoms). These times do not include the time to transfer the data, which is about the same for all the proposed methods. Metis requires about an order of magnitude more time than hashing (both forms). Between the two hashing methods, approximate hashing is about 20% faster, since it has a faster hashing function. Round robin and URL sorted require almost no time, since there is no processing required. (the data set is already sorted).

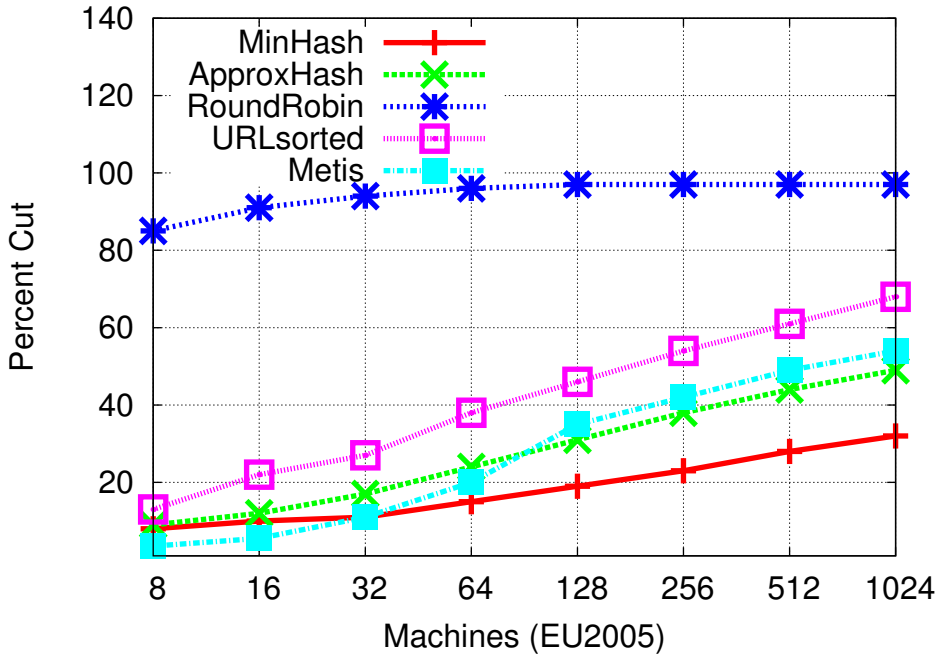


(a) Percent cut

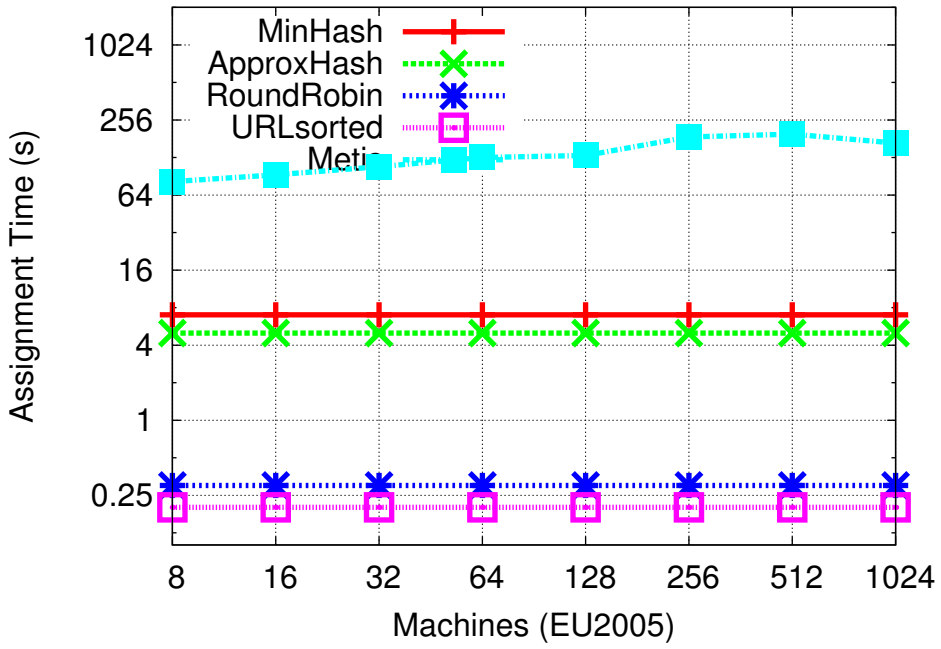


(b) Execution time (s)

Figure 8.2: The percent cuts (top) and assignment times (bottom) for partitioning the *UK2005* data set.



(a) Percent cut



(b) Execution time (s)

Figure 8.3: The percent cuts (top) and assignment times (bottom) for partitioning the *EU2005* data set.

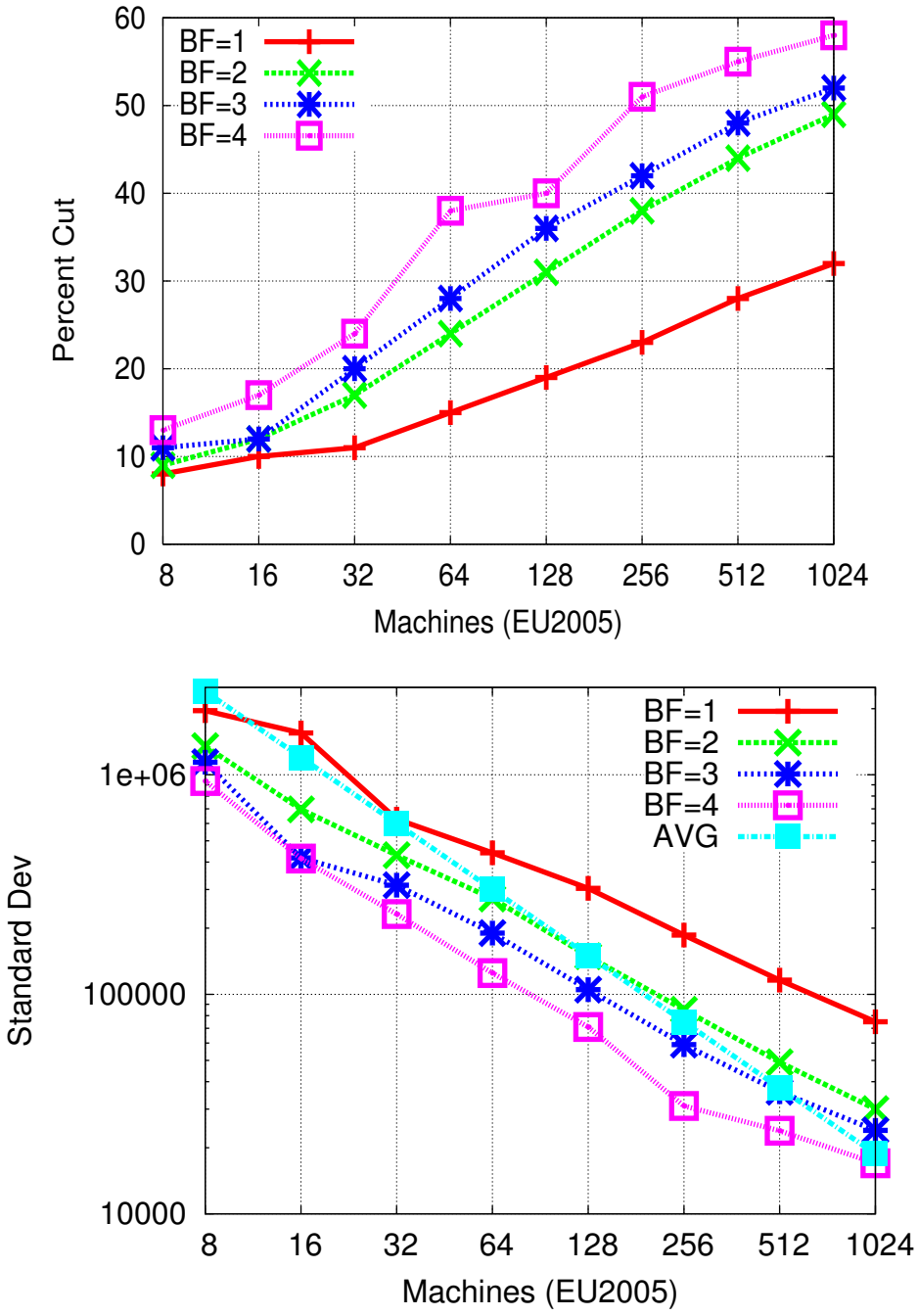


Figure 8.4: Percent cuts and the load imbalance as a function of the balance factor for the *EU2005* data set.

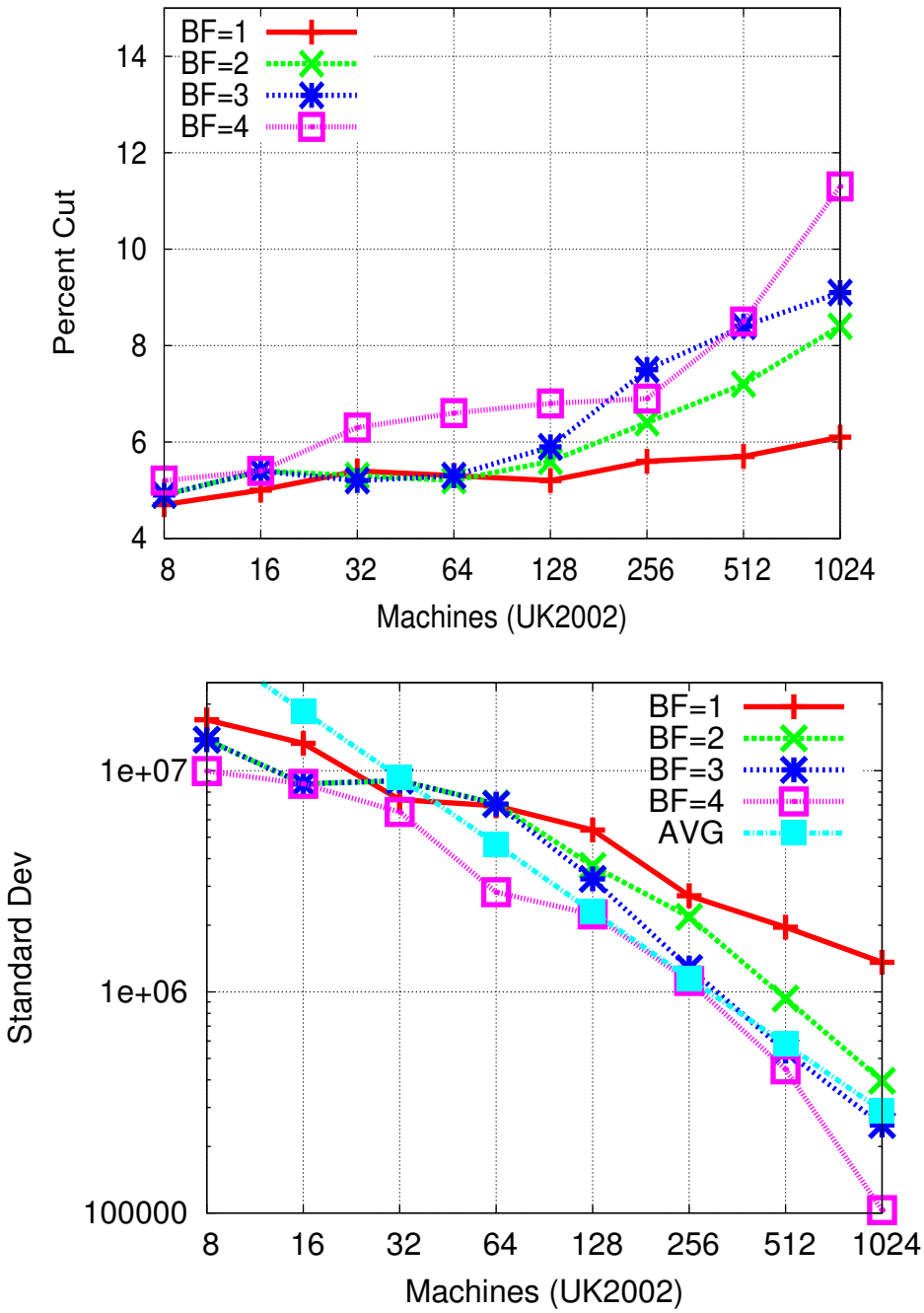


Figure 8.5: Percent cuts and the load imbalance as a function of the balance factor for the *UK2002* data set.

8.7.2 Maintaining Neighborhoods

We evaluate the placement service to maintain nearest neighbors when distributing the data set. The premise is that if similar documents are partitioned onto the same machine, then local classification will produce more accurate results with nearer neighbors, potentially lowering the need for inter machine communication. First the 20 nearest neighbors are found for 1000 random records in the data set. The Jaccard coefficient is used for similarity, and the top 20 scores are summed for each record. These sums are then summed to produce a final value. We then distribute the data, with both *min-hashing* and *round robin*. Again these 1000 records are located, and their 20 nearest neighbors are found – with the constraint that the neighbor must be on the same machine as the record in question. These values are then summed as before. We divide the sum from each partitioning by the global sum to obtain a ratio of *maintained neighbors*. Although straightforward, this experiment captures precisely what we hope to achieve, which is to group similar items onto the same machine, with little overhead.

These values are plotted for various machine cluster sizes operating on the *NLM*, *webdocs* and *EU2005* data sets in Figures 8.6 8.7, and 8.8 respectively. In all cases, local neighbors are better maintained by *min hashing*. For example, with the *webdocs* data set, the ratio drops from 62% at 8 machines to 28% on 1024 machines using the *round robin* scheme. This is understandable, as the neighborhood is essentially evenly distributed, and finding new local neighbors with high similarity is unlikely. However, using min-hashing the ratio only drops from 68% to 44% over the same range of machines. The reason is that many of the local neighbors are assigned to a machine together when they are passed to Algorithm 24. *NLM* exhibits similar

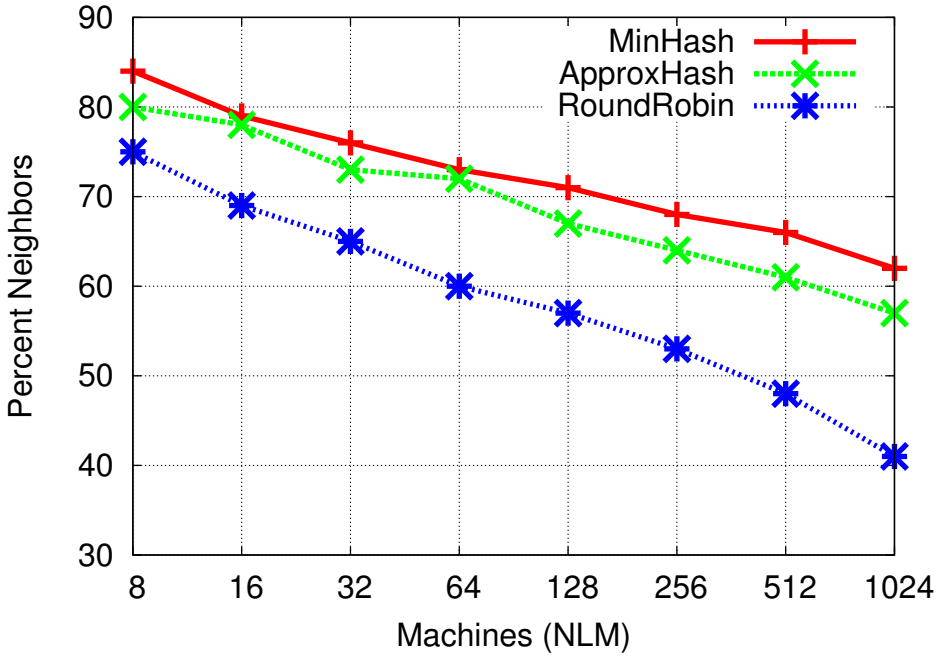


Figure 8.6: The ability of the placements algorithms to maintain neighborhoods for the *NLM* data set.

behavior, degrading from 84% to 62% using min-hashing whereas round robin degraded from 75% to 41%. The *EU2005* data set has the largest disparity between hashing and round robin, because it is a web graph and the each node has several highly similar neighbors which are found easily by hashing but lost in the round robin scheme. The performance difference between approximate hashing and min-hashing is modest. Also, unlike graph compression, the load was well distributed, with standard deviations of local sizes below 50% of the average load. Finally, we note that offline assgignment costs were in line with the graph partitioning experiments.

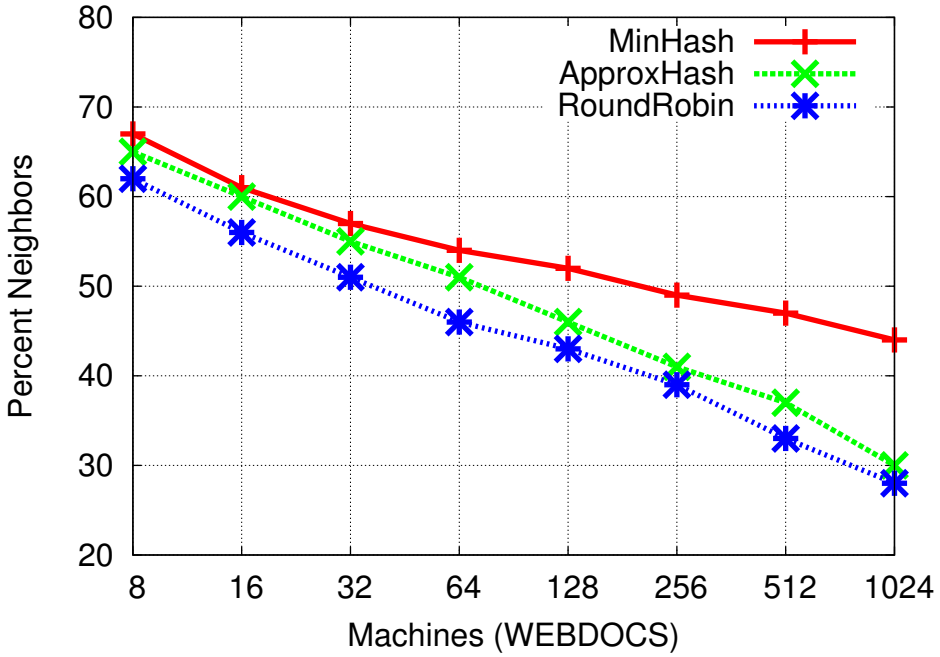


Figure 8.7: The ability of the placements algorithms to maintain neighborhoods for the *Webdocs* data set.

8.7.3 Compression Support

We evaluate our compression technique for its ability to compress the data set as we distribute it across the cluster. The procedure is as follows. We partition the data onto the cluster. Next, we compress the data on each machine, and move this compressed replication to its proper machine. We then sum the sizes of the compressed files and divide the original filesize by this sum to arrive at a compression ratio. We present the results in Figures 8.9 and 8.10, for the *ARABIC2005* and *EU2005* data sets, respectively. In the figures, *Machines=1* equates to compressing the full file on one machine. This value will be a compression maximum for the data set, since it can make maximum use of replicated itemsets. We present two curves in

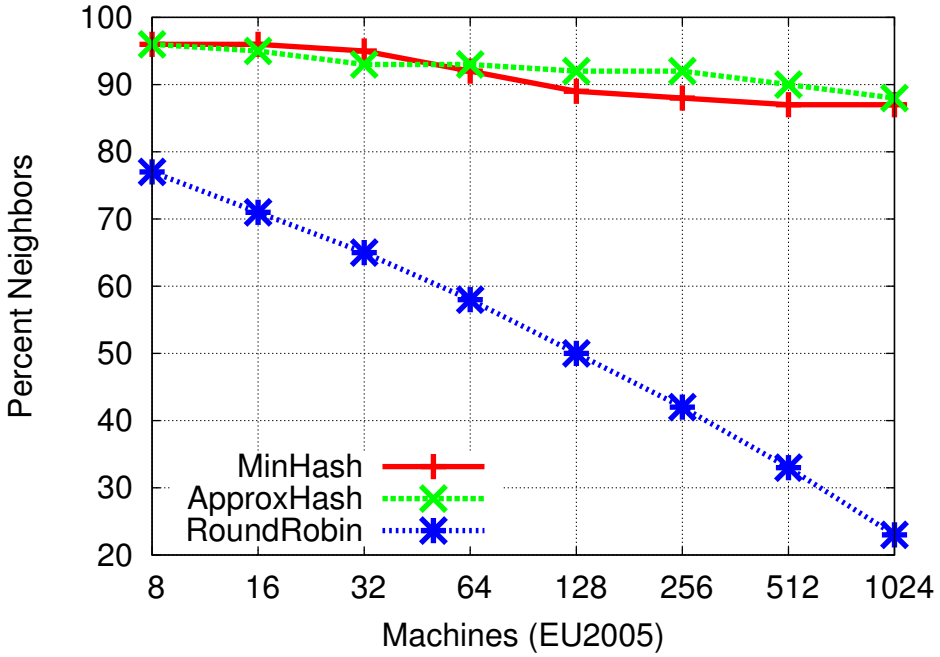


Figure 8.8: The ability of the placements algorithms to maintain neighborhoods for the *EU2005* data set.

each, which are the compression ratios when partitioning with *min-hashing* and with *round robin*.

First let us consider Figure 8.9. The maximum compression for *ARABIC2005* is a 14.3-fold reduction when compressed on a single machine. As we partition the data, it is clear that *min-hashing* maintains a larger percentage of its locality, degrading only to a ratio of 14.1-fold when distributing over 1024 machines. However, *round robin* degrades to a compression ratio of only 5.3-fold, because it does not keep similar records on the same machine.

Next we consider Figure 8.10. The maximum compression *EU2005* is a 7.3-fold reduction when compressed on a single machine. As we partition the data, again *min-hashing* maintains a larger percentage of its locality, degrading only to a ratio

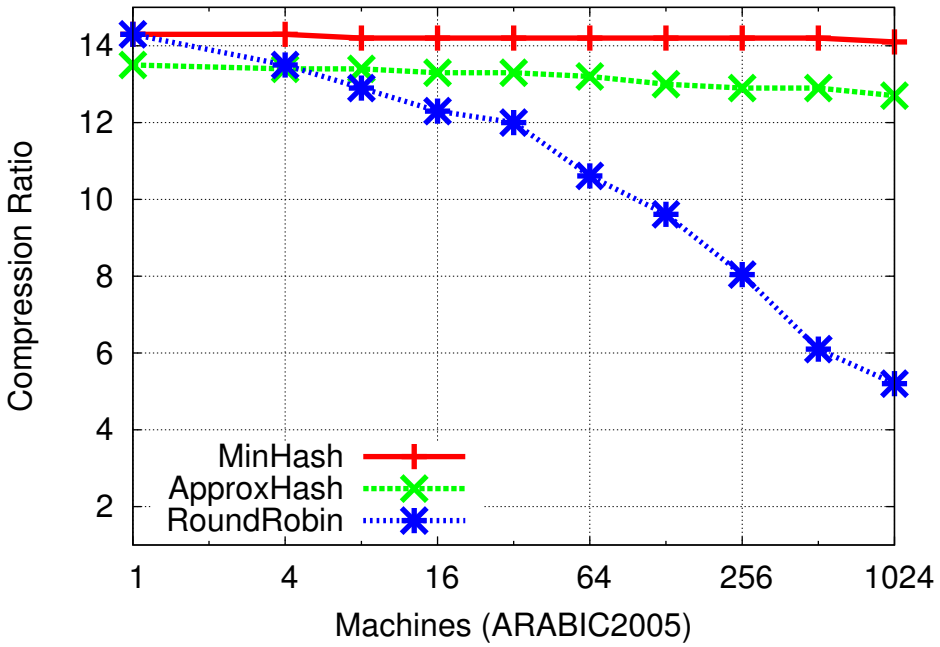


Figure 8.9: Compression ratios as a function of the cluster size for the *ARABIC2005* data set.

of 7.1-fold when distributing over 1024 machines. However, *round robin* degrades to a compression ratio of only 1.53-fold, because it does not keep similar records on the same machine. Hashing offers more than 450% more compression. Some degradation can be attributed to the small size of the input data as well. Execution times for distributed compression were presented in Section 6.5.

We also compressed the input data using *gzip (LZ77)*, which does not provide record level access but may be acceptable to many users. The *ARABIC2005* data set compressed 14.2-fold and the *EU2005* data set compressed 8.6-fold. When partitioned across the cluster, the hashing partitioning schemes provided better *gzip* compression than *round robin*, typically around 1.75-fold better.

8.7.4 Frequent Patterns

In our final set of experiments, we briefly explore the local partitions for frequent itemsets. As discussed in Chapter 3, mining for global itemsets in a distributed environment is an expensive operation. As an alternative to global mining, we can instead mine approximate patterns locally. We can achieve this by grouping similar transactions on the same machine, so that relevant patterns can be discovered locally, while maintaining the same support threshold. The benefit is that local patterns can be mined without communication of meta data or portions of the data set. To evaluate this operation, we first mine the data set at a given support on a single machine. Then, we partition the data set and mine each local partition at the same relative support. If we have increased the relative associativity in the data set, we can discover interesting patterns which would require a much smaller support threshold if mined globally.

We use the *EU2005* data set at a support of 20%. It contained 25 global closed patterns. When we partitioned it using min hashing on 8 machines, using the same relative support, we discovered 3852 closed patterns. When using round robin distribution, we discovered 132 closed patterns. In fact, discovering more patterns through hash-based placement occurred n all cases we tested.

8.8 Conclusion

In this article we engineer an API for data distribution on large data mining centers. The interface and algorithms are sufficiently general so as to support a variety of common data mining workloads, while providing the potential for improved application performance through locality sensitive placement. We demonstrate this

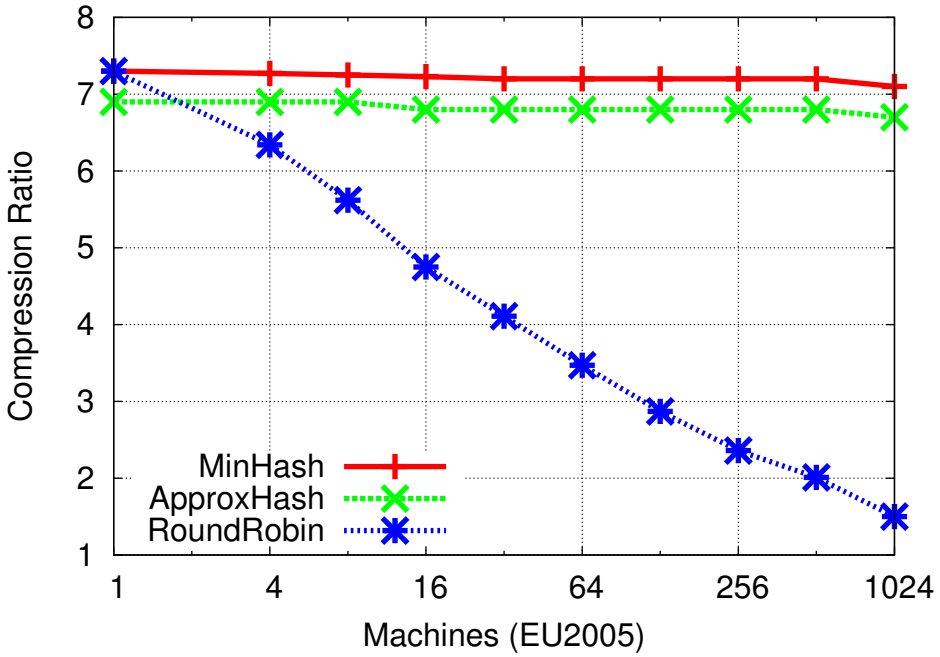


Figure 8.10: Compression ratios as a function of the cluster size for the *EU2005* data set.

potential for several key mining workloads. For classification, we demonstrate our methods maintain better local neighborhoods that other placements schemes, allowing similar records to be placed on the same machine at low cost. Also, we demonstrate the ability to lower inter-machine links when distributing large web graphs. Third, we present a distributed fault tolerant compression mechanism, for large graphs which affords comparable compression to traditional full compression schemes, while still affording node level access in the compressed form. Finally, we are in the process of implementing and testing several full applications using the proposed service and look to present these efforts in future work.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this dissertation, we have examined the data mining landscape and shown that by leveraging emerging commodity architectures, and by discovering useful patterns under a log-linear computational constraint, one can achieve highly scalable algorithms. In Chapter 3 we developed a parallel approach to global itemset mining for terascale data sets on a message-passing distributed cluster of machines. The approach lowers synchronization among nodes by requiring only one transfer round, while also reducing the data set size required at each node.

In Chapter 4, we formulated a mechanism to dynamically partition tasks onto compute nodes for CMP systems, called adaptive partitioning. The mechanism varies the size of a task at runtime, allowing work to be evenly divided among the available processing nodes. We used the technique to improve graph mining on shared memory systems, achieving over 20-fold speedups on 32 compute nodes.

We then leveraged the high floating point throughput of the Cell processor to formulate highly efficient algorithms for several key data mining kernels. For clustering, classification and outlier detection, these algorithms are up to 50 times faster than similar codes on competing technologies. More importantly, we devised a simple

evaluation process whereby algorithm designers can make an educated guess as to the viability of the Cell for other workloads.

In Chapter 6, we presented a new method to mine itemsets. We showed that this method finds more patterns with higher utility than existing strategies. Also, the method has a log-linear complexity, and inherently maintains the locations of the discovered patterns, whereas most algorithms (such as FPGrowth) are exponential and do not maintain this state.

We leverage this new technology to mine for community seeds in web graphs. In addition to discovering interesting bipartite groups, the strategy offers direct access to the seeds, supports random access to graph nodes without decompressing, and does not rely on URL ordering. Most importantly, it provides the best web graph compression to date, up to a 15-fold reduction in the size of the graph. It is efficient as well, compressing a 3 *billion* edge graph on single processor in about ten minutes on a commodity 8-core machine.

In Chapter 8 we described a data placement service designed to improve the efficiency of many of the common workloads in distributed data mining centers. The service is more general than placement schemes designed for a particular workload, but tailored to data mining more so than general-purpose data distribution methods. The high-level strategy is to place similar records on the same machine, so that common queries can be answered without global computation. The mechanisms proposed offer significant improvements in localized calculations, as well as reduced communication costs for distributed algorithms. For example, a 50% reduction in edge cuts is realized for large distributed web graphs. Also, on 1024 machines the mechanism retains almost twice as many neighbors as traditional partitioning. Finally, the mechanism

affords distributed lossless redundancy with record-level access, requiring only 7% of the disk space when compared to the original data set.

9.1 Future Directions

There are several interesting directions for future work. First, the process we developed for mining transactional data in log-linear time can be extended to mine for patterns in graph and tree data as well. A similar linear-time hashing function can be used to group similar trees, followed by an approximate one-pass projection for finding the highest utility patterns quickly. By relaxing the requirement that we must find *every* supported pattern at a given support, we can find many interesting patterns that are computationally infeasible to find using traditional methods. This extension would increase the scalability of the algorithms presented in Chapter 4.

A second direction for future work is to leverage the compressed representation of the web graph we developed in Chapter 7 for global computations. It is clear that the graph need not be compressed (besides decoding the Huffman deltas) to perform graph operations. For example, PageRank is clearly $O(E)$. By reducing the number of edges by 10-fold, we are reducing the number of required computations by 10-fold. The virtual nodes can be used as local PageRank accumulators during the iterative process. After the principle eigen vector converges, these accumulators can be distributed in a single additional pass. Since there are typically on the order of 75 iterations, the potential speedup is then about 10-fold.

We believe that web search and other large data domains are fraught with challenges requiring compression, and frequent patterns can play a role in the solution. It may be possible to use itemset mining to compress other data sets. One example

is URL data. Each URL, on average is about 80 bytes. For a 25 billion node graph, that represents about 2TB of data. Existing modified LZW methods compress this by about 4-fold. Using itemsets, one could capture correlations that string analyses often miss. For example. many image URLs end have both an *images* token as well as a *.jpg* token. Although the two tokens are not contiguous, it may be possible to catch this correlation using unordered tokens, and then use a bit mask to choose from one of only a few of the possible permutations. An alternate solution might involve approximate sequence mining instead of itemsets. Finally, we feel these compression algorithms parallelize easily. A full web graph would need this parallel formulation, and is a key direction of future work.

BIBLIOGRAPHY

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 207–218, 1993.

[2] R. Agrawal and J. Schafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–969, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.

[5] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE Press, 2006.

[6] H. Andrade, T. Kurc, J. Saltz, and A. Sussman. Decision tree construction for data mining on clusters of shared memory multiprocessors. In *HPDM: 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining*, 2003.

[7] L. Barroso, K. Gharachorloo, and F. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 3–14, 1998.

[8] S. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the 9th International Conference on Knowledge Discovery and Data mining (KDD)*, pages 478–487, 2003.

[9] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, 2000.

[10] P. Boldi and S. Vigna. The webgraph framework ii: Codes for the world-wide web. In *Technical Report 294-03*. Universitdi Milano, Dipartimento di Scienze dell'Informazione, 2003.

[11] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[12] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[13] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures in molecules. In *Proceedings of the 2nd Internation Conference on Data Mining (ICDM)*, pages 51–58, 2002.

[14] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 265–276, 1997.

[15] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[16] Andrei Z. Broder, Moses Charikarand Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[17] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[18] G. Buehrer, S. Parthasarathy, A. Nguyen, D. Kim, YK. Chen, and P. Dubey. Towards data mining on emerging architectures. In *Proceedings of the 2006 SIAM DM High Performance Data Mining Workshop*, 2006.

[19] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte data mining: An architecture conscious solution. In *Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–14, 2006.

[20] Gregory Buehrer, Srinivasan Parthasarathy, and Amol Ghoting. Out-of-core frequent pattern mining on a commodity pc. In *KDD '06: Proceedings of the 12th ACM SIGKDD international con ference on Knowledge discovery and data mining*, pages 86–95, 2006.

[21] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset mining algorithm for transactional databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 443–452, 2001.

[22] Toon Calders and Bart Goethals. Non-derivable itemset mining. *Data Mining and Knowledge Discovery*, 14(1):171–206, 2007.

[23] J. Chattratichat, J. Darlington, Y. Guo, S. Hedvall, M. Koller, and J. Syed. An architecture for distributed enterprise data mining. In *HPCN Europe*, pages 573–582, 1999.

[24] Amitabh Chaudhary, Alexander S. Szalay, and Andrew W. Moore. Very fast outlier detection in large multidimensional data sets. In *ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 2002.

[25] Thomas Chen, Ram Raghavan, Jason Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementation: A performance view. In *IBM DeveloperWorks*, http://www-128.ibm.com/developerworks/power/library/pa-cellperf/, 2005.

[26] A. Cheung and A. Reeves. High performance computing on a cluster of workstations. In *Proceedings of the Symposium on High Performance Distributed Computing (HPDC)*, pages 152–160, 1992.

[27] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 31–42, 1996.

[28] D. Cheung, K. Hu, and S. Xia. Asynchronous parallel algorithm for mining association rules on shared-memory multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 279–288, 1998.

[29] F. R. K. Chung. Spectral graph theory. 1997.

[30] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Science*, 55(3):441–453, 1997.

[31] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support pruning. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):64–78, 2001.

[32] Shengnan Cong, Jiawei Han, Jay Hoeflinger, and David 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*, pages 255–265, 2005.

[33] Diane J. Cook, Lawrence B. Holder, Gehad Galal, and Ron Maglothin. Approaches to parallel graph-based knowledge discovery. *Journal on Parallel Distributed Computing*, 61(3):427–446, 2001.

[34] T.M. Cover and P.E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[35] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, pages 229–241, 1999.

[36] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 461–470, 2007.

[37] Giuseppe Di Fatta and Michael R. Berthold. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on High Performance Computational Biology*, 17(8):773–785, 2006.

[38] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data mining (KDD)*, pages 150–160, 2000.

[39] G. W. Flake, S. Lawrence, C. L. Giles, and F. Coetzee. Self organization of the web and identification of communities. *IEEE Computer*, 35(3):66–71, 2002.

[40] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–588, 2005.

[41] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *HYPERTEXT*, pages 224–235, 1998.

[42] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of 31st International Conference on Very Large Data Bases (VLDB)*, pages 721–732, 2005.

[43] B. Goethals and M. Zaki. Advances in frequent itemset mining implementations. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

[44] A. Goinis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*, pages 518–529, 1999.

[45] K. Gouda and M. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 163–170, 2001.

[46] G. Grahne and J. Zhu. Mining frequent itemsets from secondary memory. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 91–98, 2004.

[47] R. Grossman, S. Bailey, A. Ramu, B. Malhi, and A. Turinsky. The preliminary design of papyrus: A system for high performance, distributed data mining over clusters, meta-clusters and super-clusters. In *Advances in Knowledge Discovery and Data Mining*, pages 37–43, 2000.

[48] M. Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[49] Valerie Guralnik and George Karypis. Dynamic load balancing algorithms for sequence mining. In *Univeristy of Minnesota Technical Report TR 00-056*, 2001.

[50] E. Han, G. Karypis, and V. Kumar. Scalable parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):352,377, 2000.

[51] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[52] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1–12, 2000.

[53] Heiko Hofer, Christian Borgelt, and Michael R. Berthold. Large scale mining of molecular fragments with wildcards. *Intelligent Data Analysis*, 8(5):495–504, 2004.

[54] L. Holder, D. Cook, and S. Djoko. Substructure discovery in the subdue system. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases (KDD)*, pages 169–180, 1994.

[55] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 3rd Internation Conference on Data Mining (ICDM)*, pages 549–552. IEEE Press, 2003.

[56] P. Indyk and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual Symposium on Theory of Computing*, pages 604–613, 1998.

[57] S. Jaroszewics and D. Simovici. Interestingness of frequent itemsets using bayesian networks as background knowledge. In *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 178–186, 2004.

[58] A. Javed and A. Khokhar. Frequent pattern mining on message passing multiprocessor systems. *Journal on Distributed and Parallel Databases*, 16:1–14, 2004.

[59] R. Jin and G. Agrawal. An efficient association mining implementation on cluster of smps. In *Workshop on Parallel and Distributed Data Mining (PDDM)*, pages 156–156, 2001.

[60] R. Jin and G. Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of SIAM International Conference on Data Mining (SDM)*, 2001.

[61] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.

[62] R. Jin and G. Agrawal. An algorithm for in-core frequent itemset mining on streaming data. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 210–217, 2005.

[63] M. Joshi, G. Karypis, and V. Kumar. Scalparc: A new scalable and efficient parallel classification algorithm for mining large data sets. In *Proceedings of the International Parallel Processing Symposium (IPPS)*, pages 573–579, 1998.

[64] H. Kargupta, B. Park, D. Hershberger, and E. Johnson. Collective data mining: A new perspective toward distributed data analysis. In *Advances in Distributed and Parallel Knowledge Discovery, Kargupta and Chan ed.*, 2000.

[65] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[66] J. Kim, X. Qin, and Y. Hsu. Memory characterization of a parallel data mining workload. In *Proceedings of the Workshop on Workload Characterization (WWC)*, pages 60–60, 1999.

[67] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 48(5):604–632, 1999.

[68] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *Computer Networks*, pages 1481–1493. Elsevier Science, 1999.

[69] D. Kunzman, G. Zheng, E. Bohm, and L. Kale. Charm++, offload api, and the cell processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism at PACT*, 2006.

[70] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the Internation Conference on Data Mining (ICDM)*, pages 313–320, 2001.

[71] G. Liu, H. Lu, Y. Xu, and J. Yu. Ascending frequency ordered prefix-tree: Efficient mining of frequent patterns. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 65–72, 2003.

[72] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[73] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB)*, pages 346–357, 2002.

[74] E. Markatos and T. Leblanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distriuted Systems*, 5(4):379–400, 1994.

[75] G. Martin, A. Unruh, and S. Urban. An agent infrastructure for knowledge discovery and event detection. In *Microelectronics and Computer Technology Corporation Technical Report MCC-INSL-003-99*, 1999.

[76] Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. Parallel mining for frequent fragments on a shared-memory multiprocessor -results and java-obstacles-. In Mathias Bauer, Alexander Kroner, and Boris Brandherm, editors, *LWA 2005 - Beitrage zur GI-Workshopwoche Lernen, Wissensentdeckung, Adaptivitat*, pages 196–201, 2005.

[77] Thorsten Meinl, Marc Wörlein, Ingrid Fischer, and Michael Philippsen. Mining molecular datasets on symmetric multiprocessor systems. In IEEE, editor, *Proceedings of the 2006 IEEE International Converence on Systems, Man and Cybernetics*, pages 1269–1274, 2006.

[78] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the 10th International Conference on Knowledge Discovery and Data mining (KDD)*, pages 647–652, 2004.

[79] M. Otey, C. Wang, S. Parthasarathy, A. Veloso, and W. Meira. Mining frequent itemsets in distributed and dynamic databases. In *Proceedings of the International Conference on Data Mining (ICDM)*, page 617, 2003.

[80] J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 175–186, 1995.

[81] S. Parthasarathy. Efficient progressive sampling for association rule mining. In *Proceedings of the International Conference on Data Mining (ICDM)*, page 354, 2002.

[82] S. Parthasarathy and M. Coatney. Efficient discovery of common substructures in macromolecules. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, page 362, 2002.

[83] S. Parthasarathy and S. Dwarkadas. Shared state for distributed interactive data mining applications. *Journal of Parallel and Distributed Databases*, 11(2):129–155, 2002.

[84] S. Parthasarathy and R. Subramonian. Facilitating data mining on a network of workstations. In *Advances in Distributed and Parallel Knowledge Discovery, Kargupta and Chan ed.*, 2000.

[85] S. Parthasarathy, M. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and iterative sequence mining. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 251–258, 1999.

[86] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems Journal*, 2001.

[87] Srinivasan Parthasarathy, Mohammed Javeed Zaki, and Wei Li. Memory placement techniques for parallel association mining. In *Knowledge Discovery and Data Mining*, pages 304–308, 1998".

[88] Andreas L. Prodromidis, Philip K. Chan, and Salvatore Stolfo. Meta-learning in distributed data mining systems: Issues and approaches. In *Advances in Knowledge Discovery and Data Mining*, 2000.

[89] John Punin, Mukkai Krishnamoorthy, and Mohammed J. Zaki. Logml – log markup language for web usage mining. In *WEBKDD Workshop: Mining Log Data Across All Customer TouchPoints (with SIGKDD01)*, pages 273–294, 2001.

[90] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The link database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference*. IEEE Press, 2002.

[91] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 432–444, 1995.

[92] Assaf Schuster and Ran Wolff. Communication-efficient distributed mining of association rules. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 473–484, New York, NY, USA, 2001. ACM Press.

[93] Assaf Schuster, Ran Wolff, and Dan Trock. A high-performance distributed algorithm for mining association rules. In *Proceedings of the International Conference on Data Mining (ICDM)*, page 291, 2003.

[94] Masakazu Seno and George Karypis. Lpminer: An algorithm for finding frequent itemsets using length-decreasing support constraint. In *International Conference on Data Mining*, pages 505–514, 2001.

[95] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: a scalable parallel classifier for data mining. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 544–555, 1996.

[96] Yudho Giri Sucahyo, Raj P. Gopalan, and Amit Rudra. Efficiently mining frequent patterns from dense datasets using a cluster of computers. In *Lecture Notes in Computer Science*, volume 2903, pages 233–244, 2003.

[97] Krysta Svore, Qiang Wu, Chris Burges, and Aaswath Raman. Improving web spam classification using rank-time features. In *Proceedings of the 3rd Workshop on Adversarial Information Retrieval on the Web (AirWeb)*, pages 9–16, 2007.

[98] Alexandre Termier, Marie-Christine Rousset, and Michle Sebag. Treefinder: a first step towards xml data mining. In *International Conference on Data Mining ICDM02*, pages 450–457, 2002.

[99] H. Toivenen. Sampling large database for association rules. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB)*, pages 134–145, 1996.

[100] L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8:410–421, 1979.

[101] A. Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM*, 51(3):495–515, 2004.

[102] C. Wang and S. Parthasarathy. Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 31–40, 2004.

[103] Jianyong Wang, Jiawei Han, Ying Lu, and Petre Tzvetkov. Tfp: An efficient algorithm for mining top-k frequent closed itemsets. *IEEE Transactions on Knowledge and Data Engineering*, 17(5):652–664, 2005.

[104] G. Webb. Discovering significant rules. In *Proceedings of the 12th SIGKKD International Conference on Knowledge Discovery and Data Mining*, pages 434–443, 2006.

[105] G. Webb and S. Zhang. k-optimal-rule-discovery. *Data Mining and Knowledge Discovery*, 10(1):39–79, 2005.

[106] Scott White and Padhraic Smyth. A spectral clustering approach to finding communities in graphs. In *SIAM Data Mining Conference*, 2005.

[107] S. Williams, J. Shalf, L. Oliker, S. Kamil, P Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of Computing Frontiers*, 2006.

[108] Marc Wörlein, Alexander Dreweke, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. Edgar: the embedding-based graph miner. In Thomas Grtner, Gemma C. Garriga, and Thorsten Meinl, editors, *Proceedings of the International Workshop on Mining and Learning with Graphs*, pages 221–228, 2006.

[109] M. Wrlein, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *The 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 392–403, Porto, Portugal, 2005.

[110] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 International Conference on Data Mining (ICDM)*, page 721, 2002.

[111] X. Yan and J. Han. gspan: Graph-based substructure pattern mining, expanded version. In *UIUC Technical Report*, volume UIUCDCS-R-2002-2296, 2002.

[112] X. Yan, P. Yu, and J. Han. Graph indexing: A frequent structure based approach. In *In Proceedings of the ACM SIGMOD International conference on Management of data (SIGMOD), 2004*, pages 335–346, 2004.

[113] Xifeng Yan, Hong Cheng, Jiawei Han, and Dong Xin. Summarizing itemset patterns: a profile-based approach. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 314–323, 2005.

[114] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal, special issue on Unsupervised Learning*, 2001.

[115] M. Zaki, C. Ho, and R. Agrawal. Parallel classification for data mining on shared memory multiprocessors. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 198–205, 1999.

[116] M. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proceedings of SIAM International Conference on Data Mining (SDM)*, pages 457–473, 2002.

[117] M. Zaki, S. Parthasarathy, , M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data mining (KDD)*, pages 283–296, 1997.

[118] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 1997.

[119] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.

[120] Mohammed J. Zaki. Parallel and distributed association mining: A survey. 7(4):14–25, 1999.

[121] Mohammed J. Zaki, Vinay Nadimpally, Deb Bardhan, and Chris Bystroff. Predicting protein folding pathways. *Bioinformatics*, 20(1):386–393, 2004.

[122] Osmar R. Zaane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *Proceedings of the International Conference on Data Mining (ICDM)*.

[123] J. Zhou, J. Cieslewicz, K. Ross, and M. Shah. Improving database performance on simultaneous multhithreading processors. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 49–60, 2005.

[124] Feida Zhu, Xifeng Yan, Jiawei Han, Philip S. Yu, and Hong Cheng. Mining colossal frequent patterns by core pattern fusion. In *Proceedings of the International Conference on Data Engineering*, pages 706–715, 2007.

[125] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3), 1977.