HIGH PERFORMANCE METHODS FOR

FREQUENT PATTERN MINING

by

LAN VU

B.A., University of Economics HCMC, 2004

B.A., University of Social Sciences & Humanities HCMC, 2005

M.S., University of Colorado Denver, 2009

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Computer Science and Information Systems

2014

UMI Number: 3667246

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 3667246

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

© 2014

LAN VU

ALL RIGHTS RESERVED

This thesis for the Doctor of Philosophy degree by

Lan Vu

has been approved for the

Computer Science and Information Systems Program

by

Gita Alaghband, Advisor

Tom Altman, Chair

Michael Mannino

Ilkyeun Ra

Tam Vu

November 19th 2014

Vu, Lan T (Ph.D., Computer Science and Information Systems)

High Performance Methods for Frequent Pattern Mining

Thesis directed by Professor Gita Alaghband

ABSTRACT

Current Big Data era is generating tremendous amount of data in most fields such as business, social media, engineering, and medicine. The demand to process and handle the resulting "big data" has led to the need for fast data mining methods to develop powerful and versatile analysis tools that can turn data into useful knowledge. Frequent pattern mining (FPM) is an important task in data mining with numerous applications such as recommendation systems, consumer market analysis, web mining, network intrusion detection, etc. We develop efficient high performance FPM methods for large-scale databases on different computing platforms, including personal computers (PCs), multi-core multi-socket servers, clusters and graphics processing units (GPUs). At the core of our research is a novel self-adaptive approach that performs efficiently and fast on both sparse and dense databases, and outperforms its sequential counterparts. This approach applies multiple mining strategies and dynamically switches among them based on the data characteristics detected at runtime. The research results include two sequential FPM methods (i.e. FEM and DFEM) and three parallel ones (i.e. ShaFEM, SDFEM and CGMM). These methods are applicable to develop powerful and scalable mining tools for big data analysis. We have tested, analysed and demonstrated their efficacy on selecting representative real databases publicly available at Frequent Itemset Mining Implementations Repository.

The form and content of this abstract are approved. I recommend its publication.

Approved: Gita Alaghband

I dedicate this thesis to my parents, my daughter Anh, my husband Thanh and my advisor Prof. Gita Alaghband who gave me unconditional love and support in every step of my way.

ACKNOWLEDGEMENTS

Working toward a PhD has been not only challenging but also the most exciting journey of my life. I have been lucky to receive so much love, supports, mentorships and encouragements from many people and organizations.

The most important person who inspired my PhD study is my academic advisor, Prof. Gita Alaghband. She has opened the door and walked me through to the long, difficult but beautiful journey of becoming a researcher. Her broad and in-depth expertise, unlimited working energy, unconditional support and especially, the love for her students gave me solid background on my research domain, brought me a lot of opportunities to explore the research world in most every aspects and encouraged me on my daily research for the greatest results that I present in this dissertation. From the deepest place of my heart, I really love her, be grateful for what she has done for me, and always feel that I am fortunate to have her as my PhD thesis advisor.

I would like to thank the thesis committee members Prof. Tom Altman, Prof. Ilkyeun Ra, Prof. Michael Mannino and Prof. Tam Vu. They are among my greatest professors that I have had opportunities to be their student and to learn from their valuable knowledge that I have applied to my study on high performance methods for frequent pattern mining. I believe when they read though this dissertation, they will find some parts of the knowledge that I have learned from them during my graduate study. I also would like to thank Dr. Beilei Xu for many meaningful career advices that have given me the visions for the career path that I have selected.

My dream of doing PhD would never come true without generous supports from many organizations. I would like to give my special thanks to the Department of

Computer Science & Engineering, College of Engineering & Applied Science, and University of Colorado Denver (UCD) for important financial supports of my study. In addition, I would like to acknowledge Computing Research Association (CRA-W), Anita Borg Institute, MIT, VMware, Microsoft, Xerox, Purdue University, Graduate School at UCD, NFS, ACM and the Altman family for numerous scholarships, travel awards, mentorships, etc. They have brought to me many opportunities to travel, meet and learn from researchers, professors, professionals, and talented friends from top-north universities and companies around the United States and the world. These wonderful learning experiences have helped me to sharpen my study plan and goals.

I keep the final words for my family for their love, supports and encouragements. For Anh, my lovely daughter who energizes my daily work and life, I am thankful for her non-stop asking about my graduation day that gave the strong motivation on this rough journey. I hope that she will know all that I have done and will do are for her. I have been always grateful my mother, Oanh, who nourished my interest in academic work since my childhood and my father, Tuan, who brought me the love of computers since I was twelve. Playing and working with a lot of exciting software including Turbo Pascal that he installed on my first personal computer inspired me to select the computer and information sciences for my college study. My interest in doing research started at college when I joined several research projects with a special partner, Thanh, who is my husband now. The experiences that we shared during that time motivated me to continue studying at graduate level. As a life partner, he has brought countless encouragements and supports that helped me keep moving on with my study.

TABLE OF CONTENTS

Chapter

1. Introduction 1

1.1 Knowledge Discovery from Databases 3

1.2 Data Mining 5

1.3 Mining Frequent Patterns and Association Rules 6

1.4 Computer Architectures for Sequential and Parallel Frequent Pattern Mining 7

1.5 Research Motivation 10

1.5.1 Challenges of Frequent Pattern Mining 10

1.5.2 Motivation 11

1.6 Research Contributions 11

1.6.1 FEM and DFEM Algorithms 13

1.6.2 ShaFEM Algorithm 14

1.6.3 SDFEM Algorithm 14

1.6.4 CGMM Algorithm 15

1.7 Dissertation Overview 16

2. Sequential Frequent Pattern Mining Based on Data Characteristics 17

2.1 Introduction 17

2.1.1 Motivation 17

2.1.2 Contributions 18

2.2 Background 19

2.3 Related Literature Review 20

2.3.1 Apriori Based Algorithms 22

2.3.2 Eclat Based Algorithms 23
2.3.3 FP-growth Based Algorithms 24
2.4 Self-adaptive FPM Approach Based on Data Characteristics 25
2.4.1 Observation and Analysis 26
2.4.2 Data Structures 27
2.4.3 The Proposed Approach 29
2.4.4 Constructing Conditional FP-tree and Bit Vectors 31
2.4.5 FEM and DFEM Algorithms 33
2.4.6 Switching Between Two Mining Strategies 34
2.4.7 Completeness of The Proposed Approach 36
2.5 FEM Algorithm 37
2.5.1 Algorithmic Description 37
2.5.2 Selecting Threshold K 39
2.6 DFEM Algorithm 42
2.6.1 Adopting Dynamic Threshold K 42
2.6.2 Algorithmic Description 45
2.7 Optimizing FEM and DFEM 47
2.8 Performance Evaluation 48
2.8.1 Experimental Setup 49
2.8.2 Execution Time Comparison 50
2.8.3 Memory Usage Comparison 52
2.8.4 Impact of Applying Two Mining Strategies 54
2.9 Conclusion 58
3. Parallel Frequent Pattern Mining on Shared Memory Multi-core Systems 59
3.1 Introduction 59

3.1.1 Motivation 59

3.1.2 Contributions 60

3.2 Background 61

3.2.1 Architecture of Multi-core Shared Memory System 61

3.2.2 Parallel Programming Models for Multi-core Shared Memory Systems. 62

3.2.3 FPM Challenges on Multi-core Shared Memory Systems 63

3.3 Related Literature Review 63

3.3.1 Tree Projection Partition Algorithm 64

3.3.2 MLPT Algorithm 64

3.3.3 FP-array Algorithm 64

3.4 ShaFEM Algorithm 65

3.4.1 Overview 65

3.4.2 Data Structures 66

3.5 Parallel XFP-Tree Construction 66

3.6 Parallel Frequent Pattern Generation 70

3.6.1 Parallel Frequent Pattern Generation Based on Data Characteristics 70

3.6.2 Switching Between Two Mining Strategies 74

3.7 Performance Evaluation 74

3.7.1 Experimental Setup 74

3.7.2 Execution Time 76

3.7.3 Speedup 79

3.7.4 Memory Usage 81

3.7.5 Sequential Performance Evaluation 84

3.7.6 Analyzing Performance Merits of ShaFEM 86

3.8 Conclusion 90

4. Frequent Pattern Mining Based on Multi-core Cluster 91
4.1 Introduction 91
4.1.1 Motivation 91
4.1.2 Contributions 92
4.2 Background 93
4.2.1 Architecture of Multi-core Cluster 94
4.2.2 Parallel Programming Models for Multi-core Cluster 95
4.3 Related Literature Review 98
4.4 SDFEM Algorithm 100
4.4.1 Overview 100
4.4.2 Parallel Projected XFP-tree Construction Stage 101
4.4.3 Parallel Frequent Pattern Generation Stage 104
4.5 Multi-level Load Balancing of SDFEM 105
4.5.1 Within-node Load Balancing 106
4.5.2 Between-node Load Balancing 108
4.6 Algorithmic Description 111
4.7 Performance Evaluation 116
4.7.1 Experimental Setup 116
4.7.2 Execution Time 117
4.7.3 Speedup 119
4.7.4 Impact of Hybrid MPI-OpenMP Programming Model 121
4.7.5 Impact of Different Load Balancing Techniques 122
4.8 Conclusion 124
5. Parallel Frequent Pattern Mining Based on GPU 126
5.1 Introduction 126

5.1.1 Motivation and Related Literature 126
5.1.2 Contributions 127
5.2 Background 128
5.2.1 GPU Architecture 128
5.2.2 CUDA Programming 130
5.2.3 Frequent Pattern Mining Using GPU 132
5.3 Prior GPU-based FPM Algorithm 132
5.3.1 CSFPM Algorithm 133
5.3.2 GPApriori Algorithm 133
5.3.3 gpuDCI Algorithm 133
5.4 New Frequent Pattern Mining Approach using a CPU-GPU Hybrid Model ... 134
5.4.1 The Proposed Multi-strategy Approach 134
5.4.2 Overview of CGMM 135
5.4.3 Switching Between the Two Mining Strategies 137
5.5 CGMM Algorithm 138
5.5.1 FP-tree construction and CPU Based Mining 138
5.5.2 GPU Based Mining 139
5.6 Performance Evaluation 145
5.6.1 Experimental Setup 145
5.6.2 Performance Evaluation 146
5.6.3 Impact of Applying Multi-Strategy Approach 149
5.6.4 Impact of Data Transfer Optimization 150
5.6.5 Impact of Threshold K 151
5.6.6 Impact of Data List Size 153
5.7 Conclusion 154

6. Conclusion 155

References …………………………………………………………………………………...158

LIST OF TABLES

Table

2-1: Execution time of three algorithms on sparse and dense databases 18

2-2: A sample dataset and its frequent items with *minsup* = 20% 20

2-3: Comparison of FEM, DFEM, Eclat, and FP-growth...34

2-4: Measurements of FEM for Kosarak (minsup=0.07%) 43

2-5: Experimential datasets and their properties 49

2-6: Descriptions of the experimential datasets 50

2-7: Peak memory usage (megabytes) of FEM, DFEM and other algorithms 53

3-1: Experimental datasets of ShaFEM 75

3-2: Test machines...76

3-3: Time comparison (sec.) of ShaFEM vs. FP-array on different hardware.................79

3-4: Performance of ShaFEM with dynamic vs. static scheduling on 12 cores................89

3-5: Performance of ShaFEM using lock vs. lock-free on 12 cores................................90

4-1: Load balancing techniques applied in SDFEM..106

4-2: Experimental datasets of SDFEM 117

4-3: Time comparision of pure MPI vs. hybrid MPI-OpenMP (60 cores) 122

4-4: Four versions of SDFEM with different load balancing techniques 123

4-5: Running time of four versions of SDFEM (120 cores) 123

5-1: Experimental datasets of CGMM...145

5-2: Speedup of CGMM vs. other sequential algorithms 148

5-3: Running time of CGMM with single mining strategy or both...............................150

5-4: Performance of CGMM with and without data transfer optimization 151

5-5: Execution time (sec.) of CGMM with different frequent pattern set list size 153

LIST OF FIGURES

Figure

1-1: The growth of world's largest databases 1

1-2: Knowledge discovery process...3

1-3: Flynn's taxonomy of computer architectures 8

1-4: Examples of different data types for FPM...10

1-5: Proposed high performance FPM methods..12

2-1: The data subsets with itemsets occurring and ending with c, e and de 26

2-2: FP-tree constructed from the dataset in Table 2-2 28

2-3: Bit Vectors constructed from the dataset in Table 2-2 29

2-4: Mining model of the proposed mining approach 30

2-5: Illustration of FP-tree and Bit Vector construction..32

2-6: The mining progress for the dataset in Table 2-2 with K=3......................................36

2-7: FEM algorithm 37

2-8: MineFPTree1 algorithm 38

2-9: MineBitVector1 algorithm...39

2-10: Running time of FEM with different values of K 41

2-11: UpdateK1 algorithm 44

2-12: DFEM algorithm 45

2-13: MineFPTree2 algorithm...46

2-14: Time comparision of FEM and DFEM with other algorithms 51

2-15: Running time of using single mining strategy vs. using both..................................55

2-16: Running time distribution of two mining strategies in DFEM 57

3-1: Architecture of multi-core shared memory system 61

3-2: Parallel construction of the global count list..68

3-3: Local FP-tree construction...69

3-4: The global shared XFP-tree 70

3-5: The frequent pattern generation model of each parallel process 71

3-6: ParallelMinePattern1 algorithm 72

3-7: MineFPTree3 algorithm 73

3-8: UpdateK2 algorithm 74

3-9: Running time comparison of ShaFEM and FP-array 78

3-10: Speedup of ShaFEM and FP-array on 12 cores relative to FP-array 1 core............79

3-11: Speedup of ShaFEM on 12 cores compared to its time on 1 core 81

3-12: Peak memory usage (megabytes) of ShaFEM and FP-array 83

3-13: Sequential running time of ShaFEM compared to sequential methods 85

3-14: Speedup of ShaFEM on 1 core compared to sequential algorithms 86

3-15: Time distribution for two mining strategies of ShaFEM.......................................88

4-1: Architecture of a cluster with 12-core nodes with dual 6-core sockets.....................95

4-2: Mapping of processes and threads to a multi-core cluster..96

4-3: Execution model of MPI program vs. hybrid MPI/OpenMP...................................97

4-4: Overview execution model of SDFEM 102

4-5: Computation of the global count by all processes (P = process; T= thread) 103

4-6: Construction of local FP-trees by each thread (T) of Process 2 (P)........................103

4-7: The project XFP-tree constructed by Process 2 104

4-8: The mining model of SDFEM within a process (T=thread)...................................105

4-9: Within-node load balancing with work sharing 107

4-10: Between-node load balancing with work stealing 109

4-11: SDFEM algorithm 111

4-12: ParallelMinePattern2 algorithm 112

4-13: MineFPTree4 algorithm 113

4-14: MineBitVector2 algorithm 114

4-15: LoadBalancing algorithm 115

4-16: UpdateK3 algorithm 116

4-17: Running time of SDFEM (from 1 to 120 cores) 118

4-18: Speedup of SDFEM (from 1 to 120 cores) 120

4-19: Running time and speedup of SDFEM when minsup varies 121

4-20: Speedup of four versions of SDFEM compared to its sequential time 124

5-1: The architecture of Nvidia's GPU 129

5-2: The execution model of CUDA 131

5-3: The overview of CGMM 136

5-4: CGMM Algorithm 138

5-5: CPUBasedMining algorithm 139

5-6: Data structures used by GPUBasedMining 140

5-7: GPUBasedMining algorithm 143

5-8: Generating pattern candidates and computing their counts on GPU 144

5-9: Running Time of CGMM vs. other sequential algorithms 147

5-10: Running time and speedup of CGMM vs. GPApriori 149

5-11 : Running time of CGMM with various values of threshold K 152

LIST OF ABBREVIATIONS AND DEFINITIONS

ARM	Association Rule Mining
CGMM	CPU & GPU based Multi-strategy Mining
Conditional Pattern Base	Sub-database consists of sets of frequent patterns given the existence of a suffix pattern
Count	Number of occurrences of an itemset given a set of transactions
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture, a parallel computing platform and programming model by NVIDIA
DFEM	Dynamic FPM algorithm, an improved version of FEM
FEM	FP-growth and Eclat based Mining
FPM	Frequent Pattern Mining
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
HPC	High Performance Computing
Itemset	A set of co-occurring items in database
IDC	International Data Corporation
KDD	Knowledge Discovery from Database
MIMD	Multiple Instruction Multiple Data
MMX	Matrix Math eXtension, a SIMD instruction set designed by Intel
Minconf	Minimum confidence threshold
Minsup	Minimum support threshold
NUMA	Non-Uniform Memory Access
OpenCL	Open Computing Language

PC	Personal Computer
SDFEM	Shared and Distributed memory based FPM algorithm
ShaFEM	Shared memory based FEM algorithm
SIMD	Single Instruction Multiple Data
SM	Stream Processor
SMP	Symmetric Multiprocessing
SMX	Next Generation Streaming Multiprocessor
SP	Stream Multiprocessor
SSE	Streaming SIMD Extension, a SIMD instruction set extension by Intel
Support	Probability of an itemset x given a set of transactions
UMA	Uniform Memory Access

1. Introduction

In the current era of information explosion, the amount of data generated in numerous research domains, such as business, social media, life science, engineering, and medicine, is rapidly growing; some examples are indicated in Figure 1-1 [1], [2], [3], [4]. A study by International Data Corporation (IDC) in 2012 predicted that the digital universe will grow up to 40,000 exabytes, or 40 trillion gigabytes by 2020 [5]. The rise of Big Data has led to the development of fast and efficient data mining methods for powerful data analysis tools that explore the tremendous amount of data and turn them into useful knowledge.

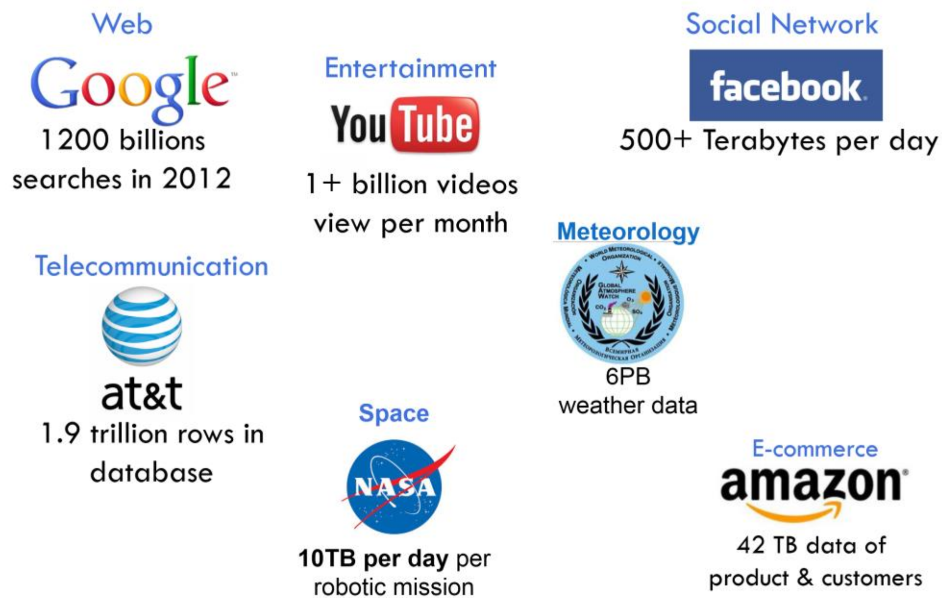


Figure 1-1: The growth of world's largest databases

In our study, we have addressed the frequent pattern mining (FPM) problem by proposing efficient high performance computing (HPC) methods to allow FPM on different database scales and various types of HPC machines, including clusters of personal computers (PCs), multi-core multi-socket servers, clusters and GPUs. The

resulting novel sequential and parallel FPM methods of our research are applicable to develop powerful data mining tools for data analysis on different computing platforms. Although our methods are neither application nor domain specific, we show that they run fast and save memory for many data types in various application domains such as web document processing, sale analysis, traffic prediction, chess game prediction and clickstream analysis as presented in the performance evaluation sections in this dissertation. It makes them promising candidates for multiple real world applications of FPM, which we review in the final chapter.

In this chapter, we introduce a general overview of concepts related to this research including knowledge discovery and data mining (Sections 1.1 and 1.2). We also present the frequent pattern mining (FPM) problem which is the target of our research and the most computationally complex and intensive step of frequent patterns and association rules discovery from database (Section 1.3). Additionally, we present the architecture of HPC machines on which the FPM task is deployed (Section 1.4). Finally, we describe the motivation and main contributions of our work (Sections 1.5 and 1.6). Chapters 2, 3, 4 and 5 describe our five FPM methods, FEM, DFEM, ShaFEM, SDFEM and CGMM designed for different computing platforms and environments. All algorithms incorporate novel approaches to FPM designs namely dynamic adaptability to data characteristics, high performance and targeted design to exploit the underlying architectural characteristics. Each chapter will include introduction and analysis, related literature review and background, design, implementation, results and performance analysis.

In this dissertation, we use the terms *method* and *algorithm; pattern* and *itemset* as well as *database* and *dataset* interchangeably.

1.1 Knowledge Discovery from Databases

Knowledge discovery from databases (KDD) is a nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data [6]. This process normally consists of multiple steps as shown in Figure 1-2, including data cleaning, data integration, data selection, data mining, evaluation and interpretation [6], [7]:

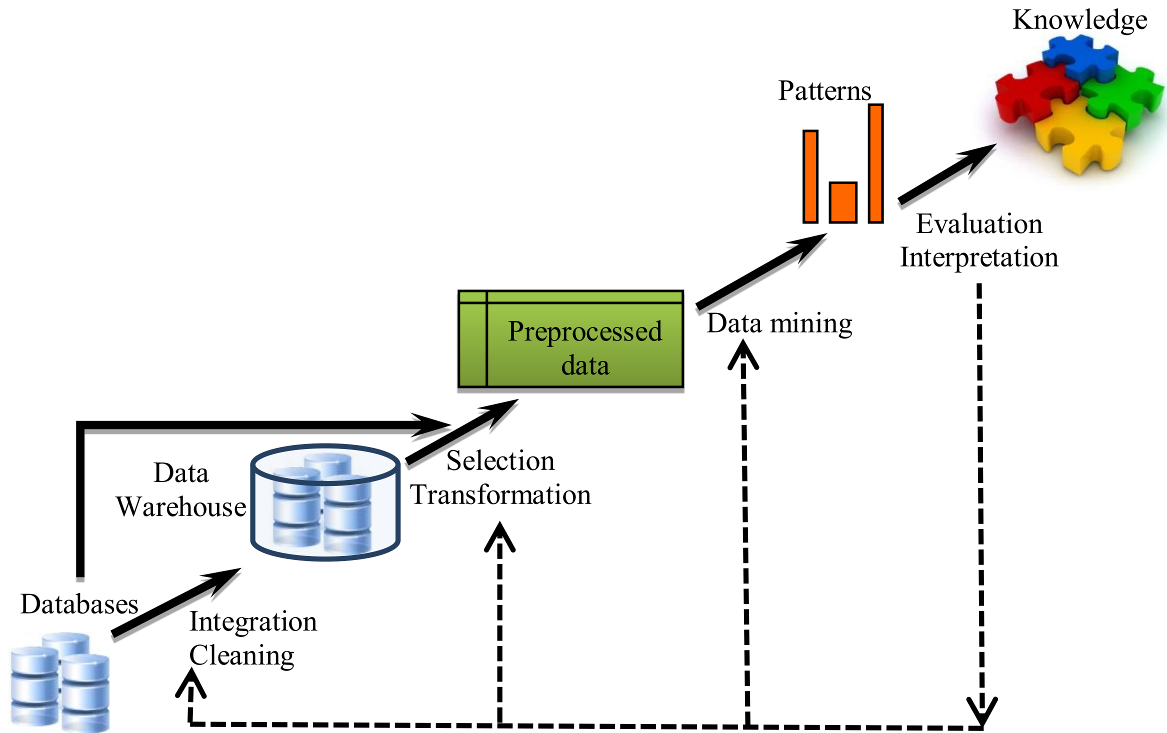


Figure 1-2: Knowledge discovery process

Data integration- In many cases, data to be mined are collected and unified from multiple data sources to provide cleansing, consistent and enough essential data for pattern discovery and to enhance the quality of extracted patterns. For large data projects,

the integrated data can be stored in a data warehouse. This step can be ignored if the collected data meet certain data integration criteria.

Data cleaning- The original databases may contain noise and inconsistent data which have significant impact on the quality of knowledge to be found by the mining task. This step is done to eliminate noise and errors when possible as well as to handle missing data and other issues of data integration such as mapping the data to a single naming convention.

Data selection- A specific data analysis task may require a portion of available data instead of the entire data source. This step queries relevant data from the databases that will be used as input for the mining step.

Data transformation- Data retrieved from database can be converted and consolidated into appropriate forms required by the mining step by operations like formatting, normalization, summarization and/or aggregation. In some cases, data dimensionality reduction methods are also applied to generate invariant representations of the data to reduce the workload and improve the quality of mined patterns. This step is sometimes performed before the data selection.

Data mining- This is the central step of the KDD process where intelligent methods are applied to extract data patterns. The selection of the data mining methods and their appropriate parameters is done based on the data mining requirements and data formats.

Evaluation and Interpretation- Pattern evaluation identifies the truly interesting patterns representing knowledge. We can utilize the assessment results to modify the computing methods used in each step or loop back to any step of the KDD process until

the generated patterns reasonably meet the application requirements. The interpretation, in addition, makes the mined patterns easier to understand by human beings by, but not limited to, using some visual forms. For many cases, the patterns are then used as the input of intelligent computing systems.

This processing flow is applied in most KDD applications with some adjustments suitable to each situation. Because most data mining algorithms from statistics, pattern recognition, and machine learning assume that data are in the main memory, the other steps in KDD ensure that the required data are available for the mining step. We now focus on the data mining step which is the heart of KDD.

1.2 Data Mining

Data mining is the essential component of KDD. It concerns the development of methods and techniques for discovering knowledge from large databases [6]. A data mining task usually handles an amount of data whose size, dimensionality and complexity are so large that use of traditional data analysis methods is impractical. The knowledge extracted from data mining is mostly in the form of patterns and can be used as the knowledge base of intelligent computing systems or to support human decision making and planning processes. Data mining involves methods in multiple disciplines including artificial intelligence, machine learning, statistics, data visualization, database technology, high performance computing and other disciplines.

In general, data mining tasks aim at discovering interesting patterns that can be used for two high level primary goals: description and prediction. Descriptive tasks characterize properties of the data in a target database. Predictive tasks perform induction on the data in order to find patterns that help with prediction of entities' behaviour.

Numerous data mining methods have been developed for these purposes and they can be grouped into five groups of data mining functionalities, which specify the kinds of patterns to be mined from data. They include (1) characterization and discrimination; (2) mining frequent patterns and associations rules; (3) classification and regression; (4) clustering analysis; and (5) outlier/abnormal analysis [8].

1.3 Mining Frequent Patterns and Association Rules

Frequent pattern mining (FPM) and association rule mining (ARM) are two important problems in data mining. FPM is a crucial part of ARM and other data mining tasks. Since their introduction for sale analysis by Agrawal et al. [8], FPM has been broadly applied in various domains with many practical applications such as market analysis, biomedical and computational biology, web mining, decision support, telecommunications alarm diagnosis and prediction, and network intrusion detection [9, 10].

Frequent Pattern Mining- FPM is used to discover many types of relationships among variables in large databases such as associations [8], correlations [11], causality [12], sequential patterns [13], episodes [14] and partial periodicity [15]. FPM is not only an essential component of ARM but also helps with data indexing, classification, clustering, and other data mining tasks [9]. It aims at searching for groups of itemsets, subsequences, or substructures that frequently co-occur in a database. A frequent pattern X is identified if the probability that X appears in the database, called *support* of X, is larger or equal to a user-specified minimum support threshold *minsup*. Because of the importance and challenges in solving this problem for large-scale databases, many studies

have focused on providing scalable FPM methods to be deployed in commercial computer systems. We present the fundamental knowledge of FPM in Chapter 2.

Association rule mining- ARM aims at discovering all interesting rules from a database that have the form: $X \rightarrow Y \mid X \cap Y = \emptyset$ where X and Y are the set of items in the database [16]. A sample association rule of retail data would be "If a customer buys milk and butter then she is 90% likely to also buy bread". Association rules are generated from a database by searching for all frequent patterns and using the two criteria, *support* and *confidence,* to identify the most important relationships. *Confidence* indicates conditional probability of finding Y given the transactions of which each contains X. In sale analysis, association rules are useful for analyzing and predicting customer behaviors. They also play an important part in shopping basket data analysis, product clustering, catalogue design and store layout [9, 10]. Other applications of ARM includes large-scale gene-expression data analysis [17], microarray data analysis [18], protein function prediction [19], risk factor prediction in business intelligence system [20], web user behaviour prediction [21].

1.4 Computer Architectures for Sequential and Parallel Frequent Pattern Mining

Depending on the scale of a FPM application and the size of its database, we can select one computing platform for a specific FPM task such as PCs, workstations, servers, clusters, supercomputers (with or without GPUs). According to Flynn's taxonomy [22], these computers can be categorized into four groups (Figure 1-3). We focus on developing FPM methods for computers of the three groups: SISD, SIMD and MIMD. MISD computers are not common and will not be considered as a platform for FPM.

- *SISD (Single Instruction, Single Data stream)*: this computer architecture allows a single instruction stream and data stream at a time. Examples of SISD computers include conventional single processor computers, such as PCs with single core processor (modern PCs have multi-core processors) or old mainframes. Although our research aims at large computer systems with multiple multi-core processors and the capability of processing multiple data elements at a time, our sequential FPM methods, FEM and DFEM, can be used for, but is not limited to, SISD machines. In sequential execution mode, FPM performance gain mainly comes from the heuristics that help with reducing the search space, the efficiency of representing and manipulating data in memory (i.e. low memory usage, better cache optimization, faster operations like bitwise operations instead of add operations).

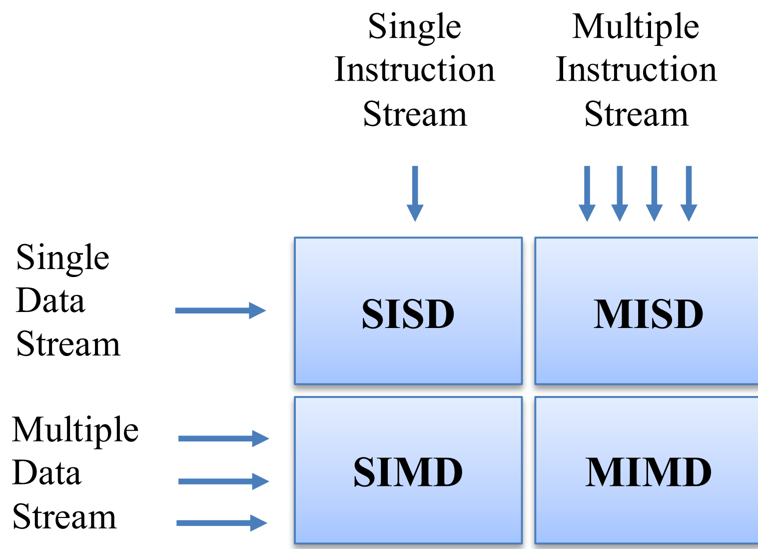


Figure 1-3: Flynn's taxonomy of computer architectures

- *MIMD (Multiple Instruction, Multiple Data streams)*- this parallel computer architecture allows simultaneous instruction execution of multiple processes on different data. Most modern computers have MIMD architecture. Examples of MIMD

include both low-end machines, such as mobile devices, PCs, and high-end ones, such as multi-core servers or clusters (i.e. machines that consist of multiple PCs or multi-core servers, called nodes, connected via high speed interconnection network) [23]. These machines, especially clusters, can potentially provide high performance for FPM and are commonly selected for deploying FPM in Big Data analysis. Because of the dominance of multi-core processor in which cores share main memory, most modern MIMD machines have shared memory architecture at some level. For example, a multi-core server, which has several multi-core chips/processors installed on a same mainboard, has MIMD shared memory architecture because its memory is accessible by all cores. Similarly, a multi-core cluster, whose nodes are multi-core machines, has distributed memory among nodes but shared memory within a node. We aim at developing parallel FPM methods that maximize the use of shared memory in MIMD systems to reduce overhead of synchronization, data communication and load balancing, which are critical issues for parallelizing FPM on large computer systems.

- *SIMD* (*Single Instruction, Multiple Data streams)*- this parallel computer architecture allows multiple data streams per single instruction stream, allowing execution of multiple identical operations on different data elements (also known as vector operation) at a time [22], [24]. Typical examples of SIMD are array processors and modern GPUs. Most modern general CPU processors have special registers integrated to support vector operations (e.g. MMX, SSE - Streaming SIMD Extensions by Intel). To fully utilize the vector computing power of SIMD processors (e.g. GPU), we design the parallel FPM method to have its data structures and computation suitable

for SIMD, particularly GPU architectures. This is a challenging problem because most existing CPU-based FPM methods use tree data structures to reduce memory usage and computation [25], [26], [27], [28], [29], [30]. This data structure, however, is not suitable for SIMD architectures.

1.5 Research Motivation

1.5.1 Challenges of Frequent Pattern Mining

FPM is usually applied to mining large databases whose size and number of distinct items are very large. For example, databases containing sale, biological and web document data (Figure 1-4) have from hundreds to millions of items and their size ranges from megabytes to terabytes. As a result, the search space for frequent patterns is extremely large. For example, given a database with 55 distinct items, there are up to 2^{55}= 36,028,797,018,963,968 different combinations, which can be frequent pattern candidates. The problem becomes more complex with larger number of items and transactions in the databases. Hence, FPM requires large computing resources in terms of both CPU time and memory usage. Developing efficient sequential and parallel methods to handle large databases becomes important to provide high performance FPM on existing computing platforms.

(a) Sale data (b) Biological data (c)Web document data

Figure 1-4: Examples of different data types for FPM

1.5.2 Motivation

Current scalable approaches for FPM have three main limitations:

1. Studies have shown that the existing mining methods (sequential and parallel) can work well only on either sparse or dense databases but not both [8], [31], [25], [26], [27], [28], [32].
2. For data-intensive problems like FPM, efficiently utilizing shared memory resources can significantly improve the mining performance. However, most parallel FPM methods were developed for distributed memory models [33], [34], [35], [36], [37], [38], even when shared memory is available in the system. The focus on distributed memory model eliminates the benefits of exploiting shared memory.
3. GPU is emerging as a ubiquitous computing device and has become an important component in HPC systems as well as data centers to support data analysis, including FPM. Although the computing power of GPU is large, designing efficient FPM algorithms for GPU is challenging due to the unpredictable workload and complex data structures of FPM. Hence, it is essential to have new efficient FPM methods that can leverage the computing power of this device.

1.6 Research Contributions

We develop novel sequential algorithms that have capability of detecting the density of data at runtime and dynamically switching to a mining strategy best suited for each subset of data being processed. We further develop parallel FPM methods that run fast and are scalable on different computer architecture platforms (e.g. cluster of PCs, multi-core shared memory servers, multi-core clusters, GPUs). These methods apply our

newly developed sequential mining method, which is described in details in Chapter 2, are able to adapt to the data characteristics for best performance on both sparse and dense databases. The significance and innovation of our research that are unique and distinguished from prior work can be highlighted in: (1) applying a multi-strategy mining approach and selecting a suitable one for each data subset upon the detection of data characteristics at runtime, (2) maximizing the use of shared memory to reduce data communication and load balancing overheads and enhance the mining performance, (3) exploiting the properties of the computing platforms on which the FPM tasks are deployed to provide mining solutions that can effectively exploit the underlying computer architectures. Figure 1-5 depicts an overview of our research that includes five new sequential and parallel FPM methods as follows:

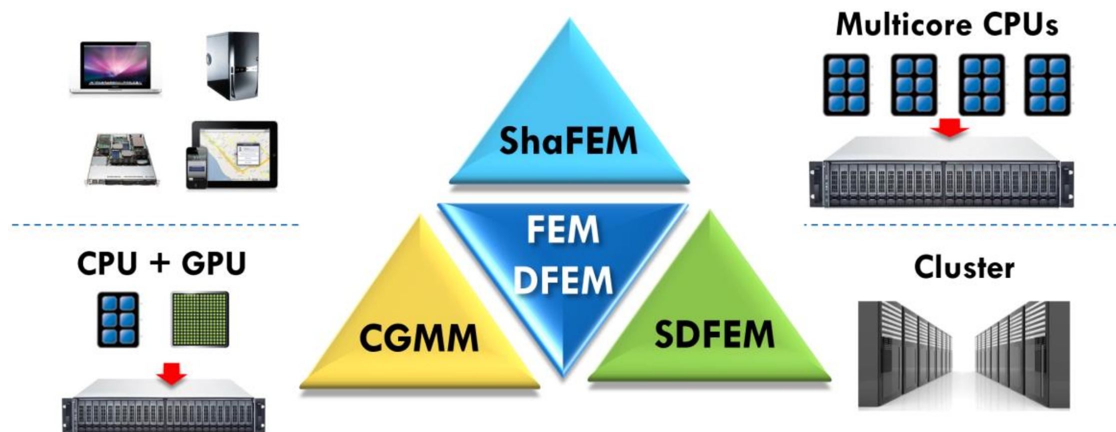


Figure 1-5: Proposed high performance FPM methods

- *FEM and DFEM*: two novel sequential FPM methods that self-adapt to data characteristics for fast performance on different data types.

- *ShaFEM*: a novel parallel FPM method for multi-core shared memory systems, which inherits mining features of DFEM and efficiently exploits the benefits of shared memory.
- *SDFEM*: a novel parallel FPM method for multi-core clusters that inherits mining features of ShaFEM and efficiently exploits the low latency communication via shared memory and the scalability of clusters consisting of multiple multi-core nodes/servers.
- *CGMM*: A novel parallel FPM method for GPU that is an enhancement of FEM and utilizes the computing power of both CPU and GPU for the FPM task.

We have presented our research results at several international conferences and some of these results in seven peer-reviewed publications [39], [40], [41], [42], [43], [44], [45] including two journals [42], [44].

1.6.1 FEM and DFEM Algorithms

Recognition that the characteristics of various data subsets within a database vary during the mining process is original to our work and the basis on which this research is built. We propose a new FPM approach that can detect the data characteristics during runtime of mining process and apply a suitable mining strategy for each data subset to generate all frequent patterns from these data. We develop two novel sequential self-adaptive FPM methods, FEM and DFEM based on the proposed methods that perform well on both sparse and dense databases. We show that FEM and DFEM outperforms (1.02 – 202.43 times faster in our experiments) and consumes less memory than many state-of-the-art methods including the well-known algorithms, namely Apriori, FP-growth and Eclat, for different types of databases. Four publications [39], [40], [41],

[42] including one journal paper [42] are among the results of our research for sequential FPM, which is presented in details in Chapter 2.

1.6.2 ShaFEM Algorithm

We develop ShaFEM, a novel parallel self-adaptive FPM method for multi-core shared memory machines (i.e. shared memory MIMD). This method is distinguished from the related works in its use of a new efficient parallel lock free method and a new data structure called XFP-tree to enhance parallelism, minimize the need for synchronization, and improve the cache utilization. In addition, ShaFEM can self-adapt to data characteristics at runtime similarly to DFEM. Our results on a 12-core multi-socket server show that ShaFEM runs 2.1 - 5.8 times faster and uses 1.7 – 7.1 times less memory on both dense and sparse databases compared to FP-array, the state-of-the-art parallel FPM method for multi-core shared memory systems. ShaFEM on 12 cores run 6.1 – 10.6 times faster than its sequential time (i.e. DFEM). This method, published in [43] and [44], is presented in Chapter 3.

1.6.3 SDFEM Algorithm

For very large-scale applications whose databases do not fit in the memory of a single server, we present a new parallel FPM method named SDFEM for multi-core clusters (i.e. shared and distributed MIMD), which runs fast and scales well on modern cluster architecture. Some features of SDFEM that make it more advanced than existing methods includes: (1) utilizes both shared and distributed memory to take advantage of low latency in-node communication; (2) applies multiple load balancing strategies to enhance the mining performance and scalability; (3) applies the FPM approach based on DFEM to enable SDFEM to work efficiently on different data types. In other words,

SDFEM is a deployment of ShaFEM on multiple nodes of a cluster with many enhancements for distributed computing environment of the cluster. In our experiments, SDFEM shows its efficiency by increasing the mining speed from 45.4 to 64.8 times on 10 twelve-core nodes (i.e. 120 cores) of a test cluster compared to its sequential execution. We describe SDFEM more details in Chapter 4. Our concepts of combining distributed and shared memory programming models for FPM was first introduced in [45]. The results of SDFEM have been submitted and is currently under review [46]. We are in preparation an extended version of the this research be submitted for publication as a journal article.

1.6.4 CGMM Algorithm

GPU with massive multi-threaded many-core architecture and SIMT(Single Instruction, Multiple Thread; SIMT model of execution on GPU is closely related to SIMD from the computational design and performance perspective) execution model makes it ideal for data processing. However, developing FPM method that can utilize GPU computing benefits is nontrivial. We design and implement CGMM, a new parallel FPM method that utilizes the computing power of both CPU and GPU for high performance. CGMM inherits FEM, uses GPU to mine dense data subsets of database and uses CPU to mine the sparse ones. The computing model of CGMM employs bit vector data structure to exploit the GPU's SIMD parallelism and to accelerate mining operations. The experimental results show that CGMM runs up to 229 times faster than six sequential algorithms on six real datasets. Additionally, CGMM runs 7.2-13.9 times faster than GPApriori, a GPU based algorithm for FPM. CGMM is a better option

compared to DFEM when mining with low *minsup* values. Chapter 5 presents the details of this method. The research results will be submitted for publication [47].

1.7 Dissertation Overview

The chapters 2, 3, 4, and 5 of this dissertation present our research including five novel FPM methods, related background, priori related works, implementation, experimental results and performance analysis of each method. Specifically, Chapter 2 presents our sequential FPM approach based on data characteristics and two resulting FEM and DFEM algorithms. We describe our parallel solutions ShaFEM for multi-core shared memory systems and SDFEM for multi-core clusters in Chapters 3 and 4 respectively. Our FPM method using hybrid GPU and CPU model is presented in Chapter 5. We introduce a brief review of FPM applications, summarize our research contributions and discuss directions for future work in Chapter 6, which is followed by a list of references of our study.

2. Sequential Frequent Pattern Mining Based on Data Characteristics

2.1 Introduction

FPM is an important data mining task for frequent pattern discovery that has been applied in numerous applications. However, FPM on databases with large number of items and transactions can lead to an explosion in computational time and memory usage. This problem is more challenging when FPM is applied with a small minimum support threshold as the requirements of the real-world applications, Google's query recommendation system for example [33]. In such a case, the search space and the number of generated frequent patterns can be very large. Therefore, developing scalable and efficient FPM methods for effective discovery of frequent patterns from databases has been a major focus of research in data mining.

2.1.1 Motivation

Several studies have shown that existing FPM methods have typically performed fast for certain types of databases. Most methods performed efficiently on either sparse or dense databases but poorly on the other [8], [25], [26], [27], [28], [31], [32], [48]. .

Table 2-1 presents the execution time of three well-known algorithms Apriori [8], Eclat [31] and FP-growth [25] on sparse and dense databases. The results shows Eclat performs best on dense data while FP-growth runs fastest on the sparse ones (the best execution times among the three algorithms are underlined). Because each algorithm performs differently on sparse and dense databases, it is difficult to select a suitable algorithm for specific applications. In addition, for commercial database systems like Oracle RDBMS, MS SQL Server, IBM DBS2 and statistical software like R, SAS and SPSS Clementine, which use FPM [49], [50], [51], [52], [53], the database characteristics

vary depending on the real applications on which they are built. For the efficiency of these systems and applications of frequent pattern mining, it is essential to have algorithms that work efficiently for most database types.

Table 2-1: Execution time of three algorithms on sparse and dense databases

Databases	**Type**	**Minsup** (%)	**Apriori** (sec.)	**Eclat** (sec.)	**FP-growth** (sec.)
Chess	Dense	20%	1924	77	89
Connect	Dense	30%	522	366	403
Retail	Sparse	0.003%	18	59	10
Kosarak	Sparse	0.08%	4332	385	144

2.1.2 Contributions

Most databases consist of both sparse and dense data portions that can only be detected during the mining process [40]. Applying single mining strategy for FPM, as existing methods do, ignores this feature and does not provide best performance for different data types. In this chapter, we present a novel high performance approach for FPM that can adapt its mining behaviors to the characteristics of databases to efficiently find all short and long patterns from various database types. The main contributions of our study include:

1. The recognition of various characteristics of databases and the fact that this characteristics may change during the mining process is our original idea. We present a novel FPM approach that can detect the data characteristics at various stages of the mining process, and select one of the two mining strategies for each data subset (Section 2.4). This new FPM approach performs fast on both sparse and dense databases. Based on this mining approach, we develop and present two

algorithms FEM (Section 2.5) and DFEM (Section 2.6). DFEM dynamically computes the best threshold value to switch its mining strategy best suitable for data subset being processed while FEM applies a fix threshold for this purpose.

2. We suggest optimization techniques for FEM and DFEM implementation in order to speed up the mining process, reduce the memory usage and optimize the I/O cost (Section 2.7).
3. We demonstrate the efficiency of FEM and DFEM in both execution time and memory usage via benchmarks of our algorithms (FEM [39] and DFEM [40]) with six other algorithms Apriori [8], Eclat [31], FP-growth [25], FP-growth* [26], FP-array [29], AIM2 [32]. The analysis of performance merits of FEM and DFEM is also presented (Section 2.8).

The remaining sections of this chapter, Sections 2.2, 2.3 and 2.9, present background, related literature review and our conclusions respectively.

2.2 Background

The FPM problem is defined as follows: Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all distinct items in the transactional database D. The *support* of an *itemset* α, a set of items, is the percentage of transactions containing α in D. A *k-itemset* α, which consists of k items from I, is frequent if *α's support* is larger or equal to *minsup,* where *minsup* is a user-specified minimum support threshold. Given a database D and a *minsup*, FPM searches for the complete set of frequent itemsets in D. For example, given the database in Table 2-2 and *minsup*=20%, the frequent 1-itemsets include *a, b, c, d* and *e* while *f* is infrequent because the *support* of f is only 11%. Similarly, *ab, ac, ad, ae, bc, bd, cd, ce, de* are frequent 2-itemsets and *abc, abd, ace, ade* are the frequent 3-itemsets.

FPM requires data to be in transactional format where each data record consists of a group of items of the same transaction. For data that originally are not in this format, data pre-processing may be required. For example, given a web document database with numerous web pages, the data of this database need some pre-processing to turn the web pages into transactions and the web page key words into items. The size of the FPM problem in such a case can be very large because the web document database can have up to billions of transactions and millions, or more, of items. The search space of itemsets is, therefore, extremely huge, making the FPM task a computational challenge. Hence, much research have focused on developing efficient and scalable heuristics FPM algorithms.

Table 2-2: A sample dataset and its frequent items with *minsup* = 20%

Transaction ID (TID)	Items	Sorted Frequent Items
1	b,d,a	a,b,d
2	c,b,d	b,c,d
3	c,d,a,e	a,c,d,e
4	d,a,e	a,d,e
5	c,b,a	a,b,c
6	c,b,a	a,b,c
7	f	
8	b,d,a	a,b,d
9	c,b,a,e,f	a,b,c,e

2.3 Related Literature Review

Most current FPM approaches [8], [25], [26], [27], [28], [31], [32], [48] utilize the *downward closure* property, in which a *k-itemset* is frequent only if its *sub-itemset*s are frequent, to significantly reduce the search space of frequent *itemsets*. First, the database D is scanned to determine all frequent items (or *1-itemsets*) in D based on the *minsup* value. After this step, only data of frequent items are used to determine the frequent

itemsets (i.e. frequent patterns). This considerably reduces the memory usage and computation by avoiding a large amount of infrequent data from being loaded into memory. Next, the frequent *(k+1)-itemsets*, initially with $k=1$, are discovered using frequent *k-itemsets* X of the previous step. For this purpose, the datasets D_X which are subsets of D and contain frequent items Y co-occurring with X ($X \cap Y = \emptyset$) are retrieved and used to determine the frequency of *(k+1)-itemsets*. Depending on the mining methods being applied, D_X can be represented in memory in many different data structures such as TID-list [31], Bitmap Vectors [10], FP-tree [25], FP-array [29], etc. or even be obtained by re-scanning the original database D from disks as in the Apriori method [8]. The characteristics of these data structures and the behaviour of their mining methods are quite different which result in different performance for a given database. For example, algorithms like Apriori [8], FP-growth [25], H-mine [27], nonordfp [28] and FP-array [29] exploit horizontal format of data and perform efficiently on sparse databases (e.g. web document data or retail data) while Eclat [31], Mafia [10], AIM2 [32] present data in vertical format and run faster on the dense ones (e.g. biological sequence data). These mining methods perform unstably on different data types as demonstrated in Table 2-1. Furthermore, the characteristics of data subsets D_X used to mine *(k+1)-itemsets* can change from very sparse to very dense as the mining task proceeds. Hence, applying a suitable mining strategy for each D_X is essential to improving the performance of FPM. We present Apriori, FP-growth, Eclat and their variants, as key FPM methods and discuss their advantages and weaknesses that motivate our research.

2.3.1 Apriori Based Algorithms

Apriori proposed by Agrawal et al. [8] is a FPM algorithm that is widely used for its simplicity. It deploys a breadth-first search to compute the *support* of k-itemsets generated from frequent (k-1)-itemsets. By utilizing the *downward closure* property that a k-itemset is frequent only if all of its sub-itemsets are frequent, Apriori achieves good performance by sharply reducing the search space. This algorithm consists of the following main steps:

1. Generate length *(k+1)*-candidate itemsets by joining two frequent *k*-itemsets if they share a common *(k-1)*-prefix.
2. Prune candidate itemsets that contain an infrequent subset
3. Count to compute the support of candidates and identify the frequent patterns by rescanning the database. Infrequent candidates are eliminated and frequent ones are reported.
4. Repeat the above steps with k=k+1 until no more frequent itemsets are discovered.

The candidate generation-and-test approach of Apriori usually requires huge computational time and memory usage when too many candidate itemsets are created. In addition, the multiple database scan strategy of Apriori is very I/O intensive when mining for large databases and/or long patterns. In spite of its disadvantages, this algorithm is considered an important milestone that opened new doors for many FPM and ARM applications. For this reason, many variants of Apriori have been proposed to address the limitations of Apriori, e.g. direct hashing and pruning (DHP) [54] , sampling technique [55], and dynamic itemset counting (DIC) [56], BitApriori [57], and Hybrid Search Based Association Rule Mining [58]. Although we do not rely on the mining approach of

Apriori for our proposed work in this chapter, its breadth first search and candidate generation-and-test is employed for our parallel GPU based mining strategy presented in Chapter 5.

2.3.2 Eclat Based Algorithms

Eclat is an efficient FPM algorithm developed by Zaki et al. [31]. This algorithm utilizes the TID-list data structure, a vertical data format, for its mining task. A TID-list of an item or itemset is a list that stores all IDs of transactions containing that item or itemset. Eclat applies the depth-first approach to search for frequent patterns and needs only one database scan. For example, Eclat reports frequent patterns following order *d, cd, cbd, acd, cb, b, ab, a* instead of the order *a, b, c, ab, cd, cd, cbd, acd* like Apriori, a the breath-first solution, does. However, two database scans can be conducted to avoid a large database loaded into memory.

The Eclat algorithm is briefly described as below:

1. Scan database D once to find all frequent items.
2. Scan database D a second time to generate the TID-lists of the frequent items.
3. Join pairs of frequent k-itemsets from the same classes of common k-prefixes to generate candidate (k+1)-itemsets. Then, intersect the TID-lists of frequent k-itemsets to compute the support of (k+1)-itemsets and to specify the frequent (k+1)-itemsets.
4. Recursively repeat step 3 until all frequent itemsets are found.

In Eclat, the *support* of an itemset is computed easily without multiple database scans as needed in the Apriori approach. Hence, the I/O cost is reduced considerably.

Eclat has been shown to be one of the best algorithms for long patterns and/or dense databases [59], [32], [48]. Although this method takes advantage of the candidate generation approach, its depth-first search order requires more infrequent itemsets generated and tested than Apriori does. As a result, Eclat's efficiency is reduced for sparse databases with short patterns where most itemsets are infrequent. Mafia [10], AIM [60], and mining using diffsets [61] are variant approaches using the vertical data format. Because of the efficiency of mining dense databases using vertical format, we utilize this mining feature as a component of our work for mining the dense data portions.

2.3.3 FP-growth Based Algorithms

FP-growth proposed by Han et al. [25] utilizes FP-tree, an extended prefix-tree that compresses all transactions of database in horizontal data format in memory, to search for the complete set of frequent patterns without the requirement of generating a large number of itemsets as Apriori and Eclat do. FP-growth involves the following steps:

1. Scan database D once to find all frequent items and generate the header table of FP-tree
2. Re-scan database D to collect the frequent items in each transaction and sort them in the frequency descending order. If the appropriate node of an item exists, its *count* is increased by one. Otherwise, a new node is inserted in the FP-tree.
3. Construct the conditional pattern base and the conditional FP-tree of each frequent item. These two data structures represent the sub dataset extracted from a FP-tree. The frequent items of new conditional FP-tree are combined with the suffix-pattern to generate the new frequent itemsets

4. Repeat step 3 on newly generated conditional FP-trees in a recursive manner until all frequent itemsets are found.

Studies have shown that FP-growth outperforms previously developed methods including Eclat and Apriori [25], [26], [27], [28], [62]. For some dense databases or mining with low minimum support, the number of frequent patterns is very large. For each frequent k-itemset, FP-growth creates s set of conditional FP-tree used to find the frequent (k+1)-itemsets. Thus, the cost of generating a large number of FP-trees results in the degradation of performance. In such cases, FP-growth does not work as well as Eclat [32], [48], [60]. The extensions of FP-growth include an array technique to reduce the FP-tree traversal time [26], H-mine [27], nonordfp [28], the use of FP-array data structure [29] and FP-growth with database partition projection [30].

Because each algorithm performs differently on sparse and dense databases, it is difficult to select a suitable algorithm for specific applications. For FPM application to perform efficiently, it is essential to have algorithms that work well for most database types.

2.4 Self-adaptive FPM Approach Based on Data Characteristics

This section lays the groundwork for our algorithms. We present our observations and analysis of the characteristics of various databases with respect to frequent patterns to motivate our new approach. We present a high-level description of our method, the relevant data structures used and the data transformation needed to switch between the two mining tasks of our approach.

2.4.1 Observation and Analysis

Most FPM methods work by generating data subsets D_x from the original database D and finding all frequent patterns from these data subsets. Studying many real databases and their characteristics, we observed that most consist of a group of items occurring much more frequently than the others. In the dense databases, most items have high frequency and appear in most transactions. In the sparse ones, the ratio of items with high frequency is considerably small compared to the total number of items. Data subsets D_X containing most frequent items have the characteristics of dense data while the ones containing less frequent items have characteristics of sparse data. For example, Figure 2-1 shows the data subsets with itemsets occurring and ending with c, e and ed (marked in black) which are extracted from Table 2-2. The data subset of c has highest density (i.e. density= (black cells / total cells)*100%) because it has two of the most frequent items a, b while the one of e, de is less dense because they contains some less frequent items like c, d. In real databases, the density of D_X can vary from low to high.

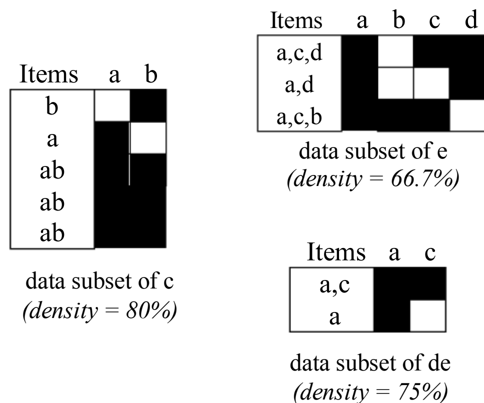


Figure 2-1: The data subsets with itemsets occurring and ending with c, e and de

In the FP-growth algorithm, the FP-tree is constructed with the nodes of the most frequent items on the top because items are added into FP-tree in a frequency descending

order. During the FP-growth mining process, conditional FP-trees (i.e. described in Section 2.4.4) are recursively constructed from parent trees. The shape of these FP-trees is usually wide for sparse databases and more compact for the dense ones. In either case, newly generated trees are much smaller than their parents because the less frequent items are trimmed out. Our studies show that the size of these trees reduces to a level where mining with alternative data structure and strategy are more efficient. At this level, the conditional FP-trees mostly consist of items with high frequencies and have the characteristics of small dense databases. Furthermore, these trees can be easily converted into vertical data structures that can be used by a mining strategy suitable for dense data. Mining these data structures are also more cache-friendly than manipulating the FP-trees with linked lists and pointers. We therefore have devised a new mining method that employs two mining strategy and dynamically switches between them based on data characteristics of D_X.

2.4.2 Data Structures

FPM is a data intensive task whose data presentation and manipulation have a large impact on mining performance. In our mining approach, we apply two main data structures: FP-tree and Bit Vector.

FP-tree is a prefix tree that compacts all sets of ordered frequent items from database into memory. This tree consists of a header table storing the frequent items with their *count*, a root node and a set of prefix sub-trees. Each node of the tree includes an *item name*, a *count* indicating the number of transactions that contain all items in the path from the root node to the current node, and a *link* to its parent node. Each linked list starting from the header table links all nodes of the same item. If two itemsets share a

common prefix, the shared part can be merged as long as the *count* properly reflects the frequency of each itemset in the database. Figure 2-2 illustrates an FP-tree constructed from the dataset in Table 2-2 where a pair <x:y> indicates *item name* and its *count*.

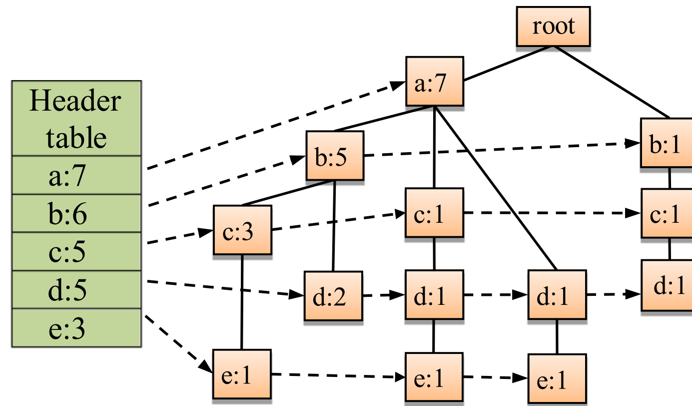


Figure 2-2: FP-tree constructed from the dataset in Table 2-2

Bit Vector is used to store data in memory using the vertical format in which data is presented column-wise and each column is associated with an item. This data structure includes *item name*, *count* and *vector of binary bits* associated with an item or itemset. The ith bit of this vector indicates if the ith transaction in the database contains that item or itemset (1: exist, 0: does not exist). For example, the dataset in Table 2-2 can be presented in five bit vectors as in Figure 2-3. The bit vector of the item f is removed because this item is infrequent. This structure not only saves memory but also enables low-cost bitwise operations for computations. In addition, our mining approach use a more compact form of these bit vectors, which reduces computation and memory cost even more.

Bit Vectors

TID	Frequent Items		a	b	c	d	e
1	a,b,d		1	1	0	1	0
2	b,c,d		0	1	1	1	0
3	a,c,d,e		1	0	1	1	1
4	a,d,e	→	1	0	0	1	1
5	a,b,c		1	1	1	0	0
6	a,b,c		1	1	1	0	0
7			0	0	0	0	0
8	a,b,d		1	1	0	1	0
9	a,b,c,e		1	1	1	0	1

Figure 2-3: Bit Vectors constructed from the dataset in Table 2-2

2.4.3 The Proposed Approach

We combine two mining strategies: (1) the first strategy applied for sparse data subsets presents data as FP-tree and uses a divide and conquer approach to generate frequent patterns; (2) the second strategy used to mine the dense data portions stores data into Bit Vectors and performs ANDing bitwise operations on pairs of vectors to specify the frequent patterns. It has been shown that while the former works better on sparse data [25], [26], [28], the latter is more suitable for dense ones [31], [32], [48]. Our approach detects the characteristics of each data subset (not whole database) and applies a suitable strategy per data subset dynamically. In general, our approach includes three main subtasks as shown Figure 2-4:

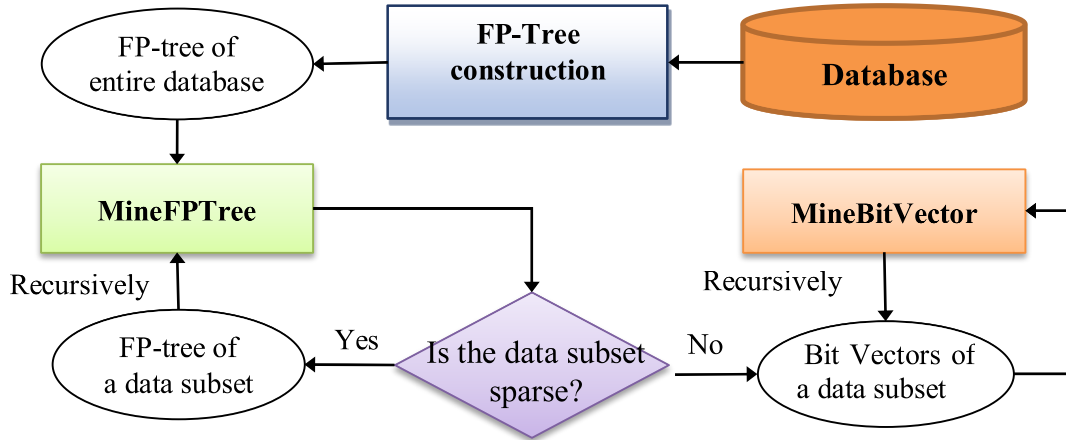


Figure 2-4: Mining model of the proposed mining approach

FP-tree construction- Database is scanned for the first time to find the frequent items and create the header table. A second database scan is conducted to get frequent items of each transaction. Then, these items are sorted and inserted into the FP-tree in their frequency descending order. During the top-down traversal for tree construction, if a node presenting an item exists, its *count* will be incremented by one. Otherwise, a new node is added to the FP-tree.

MineFPTree- This generates frequent patterns by concatenating the suffix pattern of the previous step with each item α of the input FP-tree. Then, it constructs a child FP-tree for every item α using a data subset (i.e. D_X as mentioned in the previous section, also called conditional pattern base as described in Section 2.4.4) which is extracted from the input FP-tree and consists of sets of frequent items co-occurring with the suffix pattern. The new tree is then used as the input of this recursive mining task. This mining approach explores data in the horizontal format and does not require generating a large number of candidate patterns. It has been shown to perform well on sparse databases. However, unlike the related works [25], [26], [28] that perform mining on FP-tree only, *MineFPTree* can switch to the second mining strategy when it detects that the current

data subset is dense and suitable for a second mining strategy. In this case, the data subset is converted into bit vectors and *MineBitVector* is invoked. A weight vector *w* whose elements indicate the frequency of sets in the conditional pattern base is added as the input of *MineBitVector*.

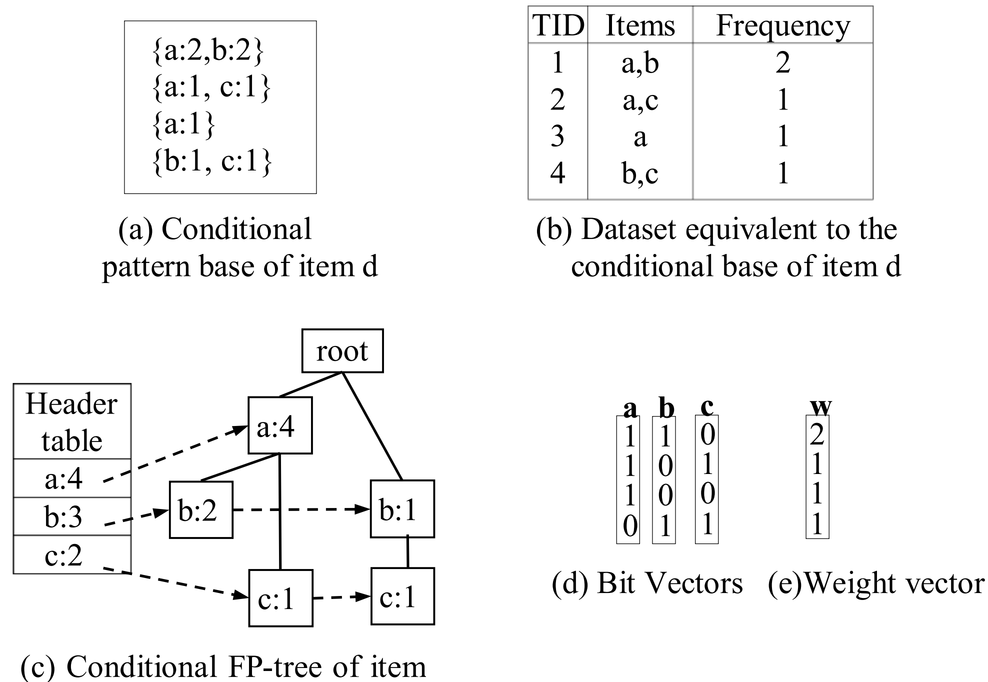
MineBitVector generates frequent patterns by concatenating the suffix pattern with each item of the input bit vector. It then joins pairs of bit vectors using logical AND operation and computes their *support* using the weight vector to specify new frequent patterns. The resulting bit vectors are used as the input of *MineBitVector* to find longer frequent patterns. The mining process continues in a recursive manner until all frequent pattern are found. The efficiency of using the vertical data format on dense data has been shown in, [10], [31], [32], [48]. *MineBitVector* is distinguished from the previous works because it uses a compact form of bit vectors where the compactness is presented in the weight vector described in next section.

In this dissertation, we use the terms *MineFPTree* and *MineBitVector* to refer to two different mining strategies as described above. These two strategies are applied in our newly developed sequential and parallel FPM methods with some adaptions based on target computing platforms. Therefore, each method has its own algorithms for *MineFPTree* and *MineBitVector* which are distinguished by the algorithm names.

2.4.4 Constructing Conditional FP-tree and Bit Vectors

Conditional pattern base- is a "sub-database" that consists of sets of frequent items co-occurring with a suffix pattern [25]. Each frequent item of a FP-tree has an equivalent conditional pattern base derived from that tree. For example, the conditional pattern base of item d, extracted from the FP-tree in Figure 2-2, consists of the 4 sets

{a:2,b:2}, {a:1, c:1}, {a:1} and {b:1, c:1} (Figure 2-5-a.) which has d as the ending item in their patterns, {a , b} occurs twice. This base is equivalent to the dataset represented in Figure 2-5b.



TID	Items	Frequency
1	a,b	2
2	a,c	1
3	a	1
4	b,c	1

(a) Conditional pattern base of item d

(b) Dataset equivalent to the conditional base of item d

(c) Conditional FP-tree of item

(d) Bit Vectors (e)Weight vector

Figure 2-5: Illustration of FP-tree and Bit Vector construction

Construction of conditional FP-tree- conditional FP-tree is a FP-tree constructed from a conditional pattern base. All frequent patterns mined from such a tree must consist of its suffix pattern. In the *MineFPTree* task, many conditional FP-trees are created using conditional pattern bases derived from the original FP-tree or their parent conditional FP-tree. Figure 2-5c shows the conditional FP-tree of item d derived from the FP-tree in Figure 2-2.

Construction of Bit Vectors and a Weight Vector- If *MineFPTree* is selected, the conditional FP-tree of item d is constructed as in Figure 2-5c. Otherwise, the bit vectors *a, b, c* and the weight vector w (Figure 2-5d and Figure 2-5e) are created instead to be

used by *MineBitVector*. The transformation to bit vectors is executed in the following steps:

1. Given a conditional pattern base with sets of n items, create n bit vectors whose size are equal to the number of sets and initialized to zero.
2. For each item in the ith set, set the ith bit of its vector to one.
3. Repeat step 2 for every available sets in the conditional pattern base.

Furthermore, each set in a conditional pattern base has a frequency value indicating the number of its occurrence. We combine all frequency values into a weight vector which is then used to compute the *support* of items or itemsets. In the given example, the weight vector is {2, 1, 1, 1} (Figure 2-5e). Thus, the data structures used in the *MineBitVector* task include a number of TID bit vectors and a weight vector *w*.

Transforming a conditional pattern base into bit vectors and a weight vector is performed without information loss. By considering a conditional pattern base as a "sub-database", we can represent it in a table with three fields TID, Items and Frequency. For example, the conditional pattern base of item d in Figure 2-5a can be presented as a dataset in Figure 2-5c. This dataset can be converted into three bit vectors (Figure 2-5d) and a weight vector (Figure 2-5e) without information loss because we can reconstruct the conditional pattern base from these bit vectors and the weight vector.

2.4.5 FEM and DFEM Algorithms

Based on the proposed approach, we introduce two FPM algorithms: FEM and DFEM where DFEM is a dynamic version of FEM. The key problem is how to decide to switch between *MineFPTree* and *MineBitVector*. We address this problem by using a

threshold K as the condition for switching between the two strategies (Section 2.4.6). FEM uses a predefined threshold K while DFEM dynamically computes K as computation proceeds and adjusts K for optimal switching decisions. Comparison of FEM, DFEM, Eclat and FP-growth is shown in Table 2-3.

Table 2-3: Comparison of FEM, DFEM, Eclat, and FP-growth

Features	FEM	DFEM	Eclat	FP-growth
Number of Database Scans	2	2	1 or 2	2
Search Order	Depth-first	Depth-first	Depth-first	Depth-first
Data Structures	FP-tree & Bit Vector	FP-tree & Bit Vector	TID-list	FP-tree
Format of Data Structures	Both	Both	Vertical	Horizontal
Threshold K	Fixed	Vary at runtime	N/A	N/A
Good Choice when Datasets	Both	Both	Dense	Sparse
Good Choice when Patterns	Both	Both	Long	Short

2.4.6 Switching Between Two Mining Strategies

Effective determination of how and when to switch between the two mining strategies is important for our mining approach to perform efficiently on different database types. During the mining process of *MineFPTree*, a large number of new FP-trees are constructed from their parent trees. A FP-tree is organized in such a way that the nodes of the most frequent items are closer to the top. The newly generated trees are much smaller than their parents because the less frequent items whose nodes are at bottom of the parent trees are removed. The size of a conditional pattern base which is used to construct new FP-tree, also reduces to a level where it contains mostly the most frequent items. In these cases, the conditional pattern base has the characteristic of a

dense dataset. Therefore, only small conditional pattern bases are considered for transforming into bit vectors and weight vector. The size of a conditional pattern base is specified by the number of sets in that base which is similar to the number of transactions in a dataset. If this size is less than or equal to a threshold K, bit vectors and a weight vector are constructed and the mining process switches to *MineBitVector*. For the FEM algorithm, the value of K is manually selected prior to execution and remains fixed while for the DFEM, this K is dynamically computed and adjusted during the mining process. If K is set to small, most of conditional pattern bases will satisfy the condition to be mined with *MineFPTree.* In contrast, when K is set to a large value, *MineBitVector* will be utilized more to mine the frequent patterns.

Figure 2-6 illustrates the mining process and the switching between the two mining strategies for the dataset in Table 2-2 when K = 3. In this case, only conditional pattern base of item d satisfies the condition to be mined with *MineFPTree* because this data subset has 4 sets whose *count* is larger than 3. All other conditional pattern bases have 3 or fewer number of sets and, therefore, are mined with *MineBitVector*. In Figure 2-6, (a),(f) is mined using *MineFPTree* while (b), (c), (d), (g), (h), (j), (k) and (l) are mined using *MineBitVector*.

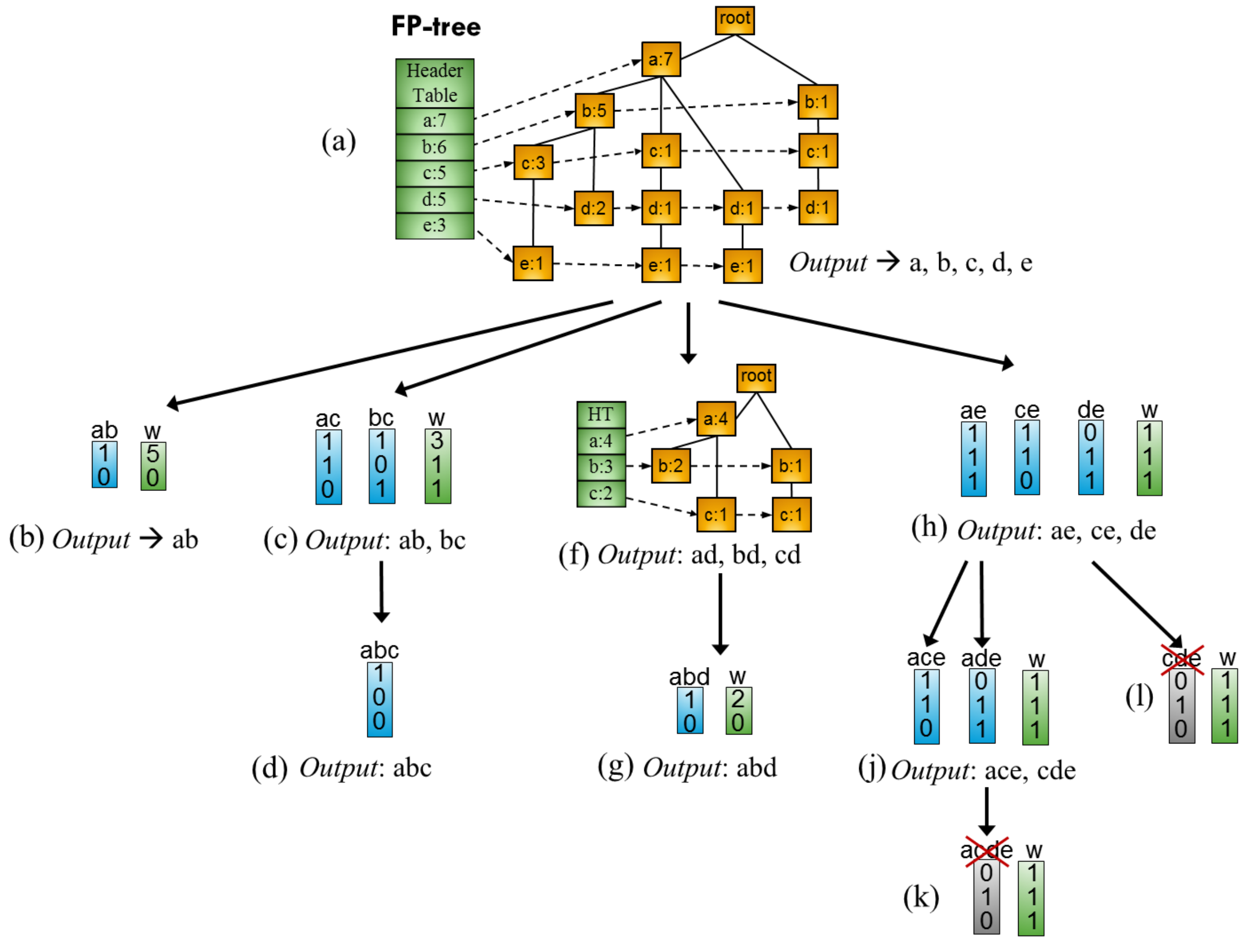


Figure 2-6: The mining progress for the dataset in Table 2-2 with K=3.

2.4.7 Completeness of The Proposed Approach

Lemma 1. Given a transactional database *D* and a minimum support threshold, a mining method combining *MineFPTree* and *MineBitVector* generates a complete set of frequent patterns in the database *D*.

Proof. If no conditional pattern base satisfies the condition for the mining process to switch to *MineBitVector*, the proposed method continues with FP-growth to generate the complete set of frequent patterns because only the *MineFPTree* task is used to find the frequent patterns in the database *D*. Given a conditional pattern base, whether it is used to create conditional FP-tree for *MineFPTree* or is transformed into TID bit vectors

for *MineBitVector*, the new frequent patterns found in either case are identical because of the integrity of data transformation and the completeness of FP-growth and Eclat. Hence, our method guarantees to generate a complete set of frequent patterns in *D*.

2.5 FEM Algorithm

2.5.1 Algorithmic Description

The FEM algorithm uses the method described in Section 2.4 and includes three sub algorithms: FEM (Figure 2-7), MineFPTree1 (Figure 2-8) and MineBitVector1 (Figure 2-9). FEM first constructs the FP-tree from the database and then calls MineFPTree1 to start searching for frequent patterns and dynamically switch to MineBitVector1 if appropriate. We recommend to use a default value *K=128* (Line 3) upon our analysis in Section 2.5.2.

FEM algorithm
Input: Transactional database *D* and *minsup* *Output*: Complete set of frequent patterns 1: Scan *D* once to find all frequent items 2: Scan *D* a second time to construct the FP-tree *T* 3: K = 128 4: Call **MineFPTree1**(*T*,∅,*minsup*)

Figure 2-7: FEM algorithm

MineFPTree1 algorithm (Figure 2-8) includes steps to perform the *MineFPTree* task described in Section 2.4. Line 6 in Figure 2-8 is a traversal of FP-tree *T* to specify the conditional pattern base of item α.The *size* in line 7 uses the number of nodes in the linked list of item α to present the size of the conditional pattern base. This size can be either computed during the execution of the task in Line 6 or during the construction of a FP-tree. Lines 8-14 show the switching between two mining tasks:

MineFPTree1 algorithm
Input: Conditional FP-Tree T, *suffix*, *minsup*
Output: Set of frequent patterns
1: **If** T contains a single path P
2: **Then For each** combination x of the items in T
3: Output $\beta = x \cup suffix$
4: **Else For** each item α in the header table of T
5: { Output $\beta = \alpha \cup suffix$
6: Construct α*'s* conditional pattern base C
7: *size* = the number of nodes in the linked list of α
8: **If** *size*>K
9: **Then** { Construct α's conditional FP-tree T'
10: Call **MineFPTree1**(T',β,*minsup*)}
11: **Else** { Transform C into TID bit vectors V
12: and weight vector w
13: Call **MineBitVector1** (V,w,β,*minsup*) }
14: }

Figure 2-8: MineFPTree1 algorithm

MineBitVector1 algorithm (Figure 2-9) performs the *MineBitVector* task described in Section 2.4. It recursively use MineBitVector1 until no new frequent patterns are found. In order to specify whether all u_j in U are identical to v_i as in Line 9, we count the number of u_j whose *support* is equal to v_i.

MineBitVector1 algorithm

Input: Bit vectors V, weight vector w, *suffix*, *minsup*

Output: Set of frequent patterns

1: Sort V in support-descending order of their items

2: **For** each vector v_i in V

3: { Output β = item of $v_i \cup suffix$

4: **For** each vector v_j in V *with* $j < i$

5: { $u_j = v_i$ AND v_j

6: sup_j = support of u_j computed using w

7: **If** $sup_j \geq minsup$ **Then** add u_j into U

8: }

9: **If** all u_j in U are identical to v_i

10: **Then For each** combination x of the items in U

11: Output $\beta' = x \cup \beta$

12: **Else If** U is not empty

13: **Then** Call **MineBitVector1**($U, w, \beta, minsup$)

14: }

Figure 2-9: MineBitVector1 algorithm

2.5.2 Selecting Threshold K

We investigate the impact of varying values of K on various databases in this section. Selecting a good value of K also depends on the minimum support specified by the user. This raises the question of how to select a good value of K without testing all possible values?. To answer this question, we conduct an experiment from which we suggest a “good” default value of K for FEM. In this experiment, we measure the performance of FEM on eight real datasets for varying values of K. The eight datasets selected for these test cases consist of a mix of four dense, three sparse and one moderate

to represent a variety of database characteristics. The detailed characteristics of these datasets are presented in Table 2-5 (Section 2.8.1). Values of *K* are selected in the range of 0-256 as multiples of 32 so that the maximum size of TID bit vectors are also multiples of 32 (4 bytes) for optimal memory utilization. Figure 2-10 indicates the running time of FEM on every dataset for varying values of *K*.

As shown in the results, FEM performs better with *K* in 32 – 128 range for seven of the datasets. For the Kosarak dataset, FEM performs best with a value of *K* in 128 – 256 range. These reflect the execution time with the selected *minsup* values. Based on extensive tests not presented here, we observed that when *minsup* varies, the range of *K* for each dataset to produce best performance also changes but it remains within the overall range of 0 – 256. For this reason, we recommend *K=128* as a default value for good performance on various databases and different *minsup* values. For *K=128*, the maximum size of a TID bit vector is 128 bits (16 bytes.) This is smaller than or equal to the size of FP-tree with many nodes where size of each node needs at least 16 bytes for item name (4 bytes), count (4 bytes), a link to parent node (4 bytes) and a link to the next node of its linked list (4 bytes). The total memory size of all TID bit vectors is therefore not greater than the number of items in the conditional pattern base multiplied by 16 bytes. This data structure requires much less memory space than an equivalent conditional FP-tree does. Furthermore, the bitwise operations on TID bit vectors will perform faster than creating and manipulating FP-trees.

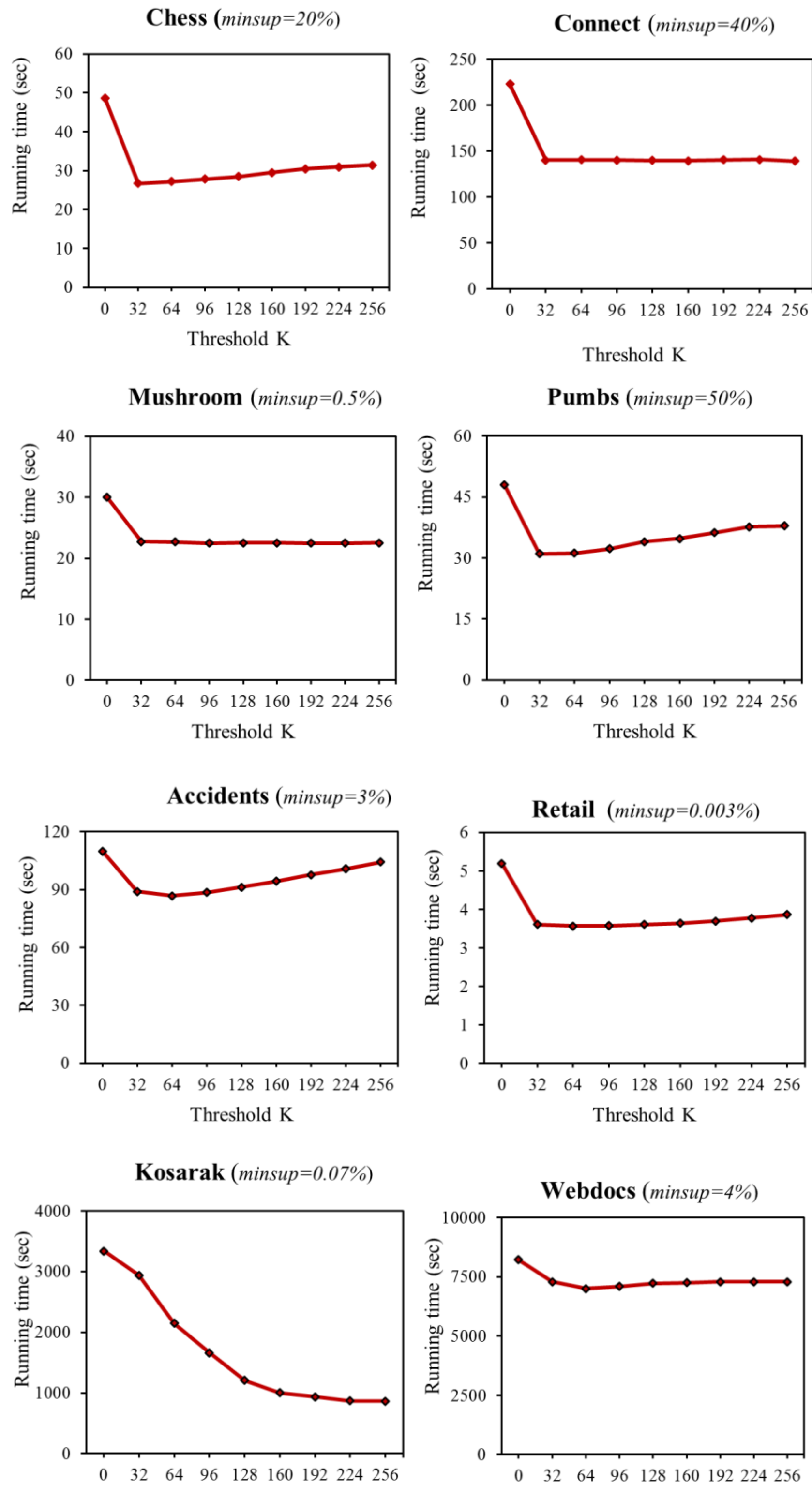


Figure 2-10: Running time of FEM with different values of K

2.6 DFEM Algorithm

DFEM is a major improvement of FEM. Unlike FEM, it computes the value of K at runtime as mining proceeds to determine a more near optimal switch point between *MineFPTree* and *MineBitVector* [40].

2.6.1 Adopting Dynamic Threshold K

In FEM, a K value *of 128* performs well on many databases. However, $K=128$ does not guarantee the best performance on all types of data. The second column in Table 2-4 shows the runtime of FEM for Kosarak dataset with different values of K and *minsup*=0.07%. As can be seen, for $K=224$, the running time of FEM is 871 seconds, significantly faster than its running time of 1206 seconds for $K=128$. This execution time difference becomes significantly larger when the minimum support threshold (*minsup*) is set to lower levels as required by many applications such as Google's query recommendation [33]. Therefore, it is important to try to find the best possible value of K dynamically as the program runs on a specific database with the required *minsup*s to gain near-optimal performance. However, finding this value of K cannot incur overhead to offset the resulting performance gain.

In Table 2-4, when K increases, the number of frequent patterns found solely by the *MineFPTree* task reduces because more of the mining workload is shifted to *MineBitVector*. Let $\{K_0, K_1, \ldots K_n\}$ be the set of K values where $K_i = K_{i-1}+32$. Denote by P_i the number of frequent patterns generated by *MineFPTree* using the threshold value K_i, and R_i the ratio P_{i-1} to P_i, defined as:

$$R_i = P_{i-1}/P_i \quad , i = 1 \ldots n \tag{2-1}$$

To study the impact of varying K_i and its relationship to R_i, we ran numerous experiments on many datasets. Our empirical study indicates that the best K_i is one satisfying:

$$R_i < 2 \ni (\nexists R_j \geq 2, \ \forall j > i) \quad (2\text{-}2)$$

In other words, FEM will perform best (near optimal) at the smallest K_i where increasing K does not result in a sharp drop in the number of frequent patterns found by *MineFPTree*.

Table 2-4: Measurements of FEM for Kosarak (minsup=0.07%)

Thres. K_i	**Running time (second)**	**# patterns by the *MineFP-tree* task (P_i)**	**Ratio R_i**
0	3341	2776266097	N/A
32	2939	1316339679	2.1
64	2146	206479285	6.4
96	1664	26795140	7.7
128	1206	2413815	11.1
160	1005	407051	5.9
192	934	86575	4.7
224	**871**	**63876**	**1.4**
256	870	58304	1.1

In the example shown in Table 2-4, the K_i value of 224, of which the running time is 871 seconds, satisfies the condition (Eq. 2-2). This result is promising. However, the challenge is that such a value of K_i can only be determined after the mining process completes and all P_i*'s* and R_i*'s* have been computed. We have developed a practical heuristics method to predict a near optimal value of K. The prediction is based on all P_i*'s*,

which are estimated dynamically at runtime, and is described in the UpdateK1 algorithm (Figure 2-11).

```
UpdateK1 algorithm
Input: NewPatterns and Size
Output:  updated value of threshold K
1:  newK = 0
2:  P[0] = P[0] + NewPatterns
3:  For i = 1 to N – 1 step 1
4:  {  If Size >i*Step
5:     {  P[i] = P[i] + NewPatterns
6:         If P[i-1] >= 2* P[i] Then newK = (i+1)*Step
7:     } Else Exit Loop
8:  }
9:  i = K/Step - 1
10: If  (i>0 AND P[i-1]  < 2*P[i] ) Then K = 0
12: If newK > K Then  K = newK
```

Figure 2-11: UpdateK1 algorithm

In this algorithm, K is the global threshold initialized to zero; N is the number of different threshold values, K_i, i=1…N, to be considered (N is set to 9 by default); *Step* is the distance between K_i and K_{i-1} (*Step* is set to 32 by default) and *P[N]* is an array. The ith element of P, P_i, stores the number of frequent patterns generated by *MineFPTree* using the threshold K_i. All P_i*'s* are initialized to zero and are regularly updated using UpdateK1 each time a new conditional pattern base is processed. K is then set to the new value *newK* that satisfies the condition (Eq. 2-2). The UpdateK1 algorithm is described in Figure 2-11; *NewPatterns* indicates the number of new frequent patterns and is equal to the number of items in C (conditional pattern base); *Size* is the size of C. Instead of all the K_i values in [1:N], we check only K_i of {0, 32, 64, … N} for two reasons: (1) to reduce

the number of computations and (2) to have a good match with most machine's word and cache block sizes because the bit vectors of *MineBitVector* are presented as arrays of 32 bit words.

2.6.2 Algorithmic Description

The DFEM algorithm uses UpdateK1 (Figure 2-11) to dynamically compute a value of K at runtime that provides a more optimal decision point to switch between the mining strategies best matching the characteristics of processed data. DFEM (Figure 2-12) consists of UpdateK1 (Figure 2-11), MineFPTree2 (Figure 2-13) and MineBitVector1 (Figure 2-8). The MineBitVector1 algorithm of DFEM is similar to that of FEM. DFEM builds the FP-tree, initializes the variables used by UpdateK1 and invokes MineFPTree2. The variables in Lines 3 must be declared in a scope that UpdateK1 can access and update. Line 6 is used to set all elements of *P[N]* to the number of frequent items in the database *D*. This step is essential for the stability of the DFEM algorithm.

DFEM algorithm
Input: Transactional database *D* and *minsup*
Output: Complete set of frequent patterns
1: Scan *D* once to find all frequent items
2: Scan *D* a second time to construct the FP-tree *T*
3: *N* = *9, Step* = 32, *K* = 0
4: Create *P*[*N*] and set all elements to zero
5: *items* = the number of frequent items in *D*
6: Call **UpdateK1**(*items*, *N**Step*)
7: Call **MineFPTree2**(*T*,∅,*minsup*)

Figure 2-12: DFEM algorithm

MineFPTree2 is similar to MineFPTree1 except that it uses UpdateK1 to regularly update K (lines 4-5, 11-13).

MineFPTree2 algorithm
Input: Conditional FP-Tree T, *suffix*, *minsup*
Output: Set of frequent patterns
1: **If** FP-tree T contains a single path P
2: { **For each** combination x of the items in P
3: { Output $\beta = x \cup suffix$ }
4: n = the number of outputs β
5: Call **UpdateK1** $(n,1)$ }
7: **Else**
8: { **For each** item α in the header table of FP-tree T
9: { Output $\beta = \alpha \cup suffix$
10: Construct α's conditional pattern base C
11: n = the number of items in C
12: *size* = the number of nodes in the linked list of α
13: Call **UpdateK1** $(n, size)$
14: **If** $size > K$ Then
15: { Construct α's conditional FP-tree T'
16: Call **MineFPTree2**$(T', \beta, minsup)$ }
17: **Else**
18: { Transform C into TID bit vectors V
19: and weight vector w
20: Call **MineBitVector1**$(V, w, \beta, minsup)$ }
21: }
22: }

Figure 2-13: MineFPTree2 algorithm

2.7 Optimizing FEM and DFEM

The mining strategies and their data structures are the most important elements that decide the performance of a frequent pattern mining algorithm. The architecture of the machine on which a frequent pattern mining program runs also has a significant impact on its runtime. It is essential to have implementation techniques that not only reduce the computational time of the CPU but also optimize the usage of cache, memory and I/O. In this section, we present a combination of optimization techniques that we have incorporated in the implementation of FEM and DFEM.

FP-tree construction- In the second database scan, FEM and DFEM pre-load the frequency descending sorted sets of frequent items into a lexicographically sorted list, a variant of the optimization suggested in [62]. One copy of similar transactions is kept with its *count*. For very large databases, the transaction list size is set at runtime to fit the available memory. We organize this list in a binary tree and maintain its order while the list grows in size. When its size limit is reached, the sets of frequent items and their *count*s are extracted from the list one by one to build the FP-tree. Therefore, the construction time of FP-tree is significantly reduced because similar itemsets are added into FP-tree only once. Moreover, the lexicographical order of the transaction list makes the FP-tree nodes most visited together to be allocated close together in memory optimizing the use of cache and speeding up the mining stage as well.

FP-tree mining task- We improve the technique proposed in [26] to implement an additional array associated with each FP-tree to pre-compute the *count* of new patterns. It helps to reduce the traversal cost of parent FP-trees when constructing the child FP-trees. The resulting performance improvement is due to maximizing locality of consistent

memory access patterns. However, for the trees with large number of frequent items, the array size is very large which consequently consumes a large amount of memory and increases the execution time. Therefore, we only enable this technique in FEM and DFEM whenever the array size does not grow beyond a predefined limit; default value is 64KB.

Memory management- For better memory utilization, large chunks of memory are allocated to store data of all FP-trees and bit vectors similar to the technique used in [26]. When all frequent patterns from a FP-tree or bit vectors and their child FP-trees or bit vectors have been found, the storage for these data structures are discarded. The chunk size is variable. This technique minimizes the overhead of allocating and freeing small pieces of data and prevents data scattered in memory.

Output processing- The most frequent output values are pre-processed and stored in an indexed table as proposed in [32]. In addition, the similar part of two frequent itemsets outputted consecutively is processed only once. This technique considerably reduces the computational time on output reporting, especially when the output size is large.

I/O optimization- Data are read into a buffer before being parsed into transactions. Similarly, the outputs are buffered and only written when the buffer is full. This technique reduces much of the I/O overhead.

2.8 Performance Evaluation

We present three experiments using eight datasets to evaluate performance and efficiency of FEM and DFEM and compare them to six other efficient algorithms. These experiments include an execution time comparison, a memory usage comparison and an

analysis of the reason for performance merits of FEM and DFEM for different types of databases.

2.8.1 Experimental Setup

Datasets: A total of eight real datasets with various characteristics and domains were selected. They include four dense, three sparse and one moderate datasets and are publicly available at the Frequent Itemset Mining Implementations Repository [59], a well-known data repository for benchmarking FPM methods. The datasets are reported in Table 2-5 and further described in Table 2-6.

Table 2-5: Experimential datasets and their properties

Datasets	Type	# Items	Average Length	# Transactions
Chess	Dense	76	37	3196
Connect	Dense	129	43	67557
Mushroom	Dense	119	23	8124
Pumsb	Dense	2113	74	49046
Accidents	Moderate	468	33.8	340183
Retail	Sparse	16470	10.3	88126
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Software: A total of eight algorithms were benchmarked: FEM, DFEM and six frequent pattern mining algorithms: Apriori [8], Eclat [31], FP-growth [25], FP-growth* [26], FP-array [29], AIM2 [32]. FEM and DFEM are implemented using our proposed methods and the optimization techniques introduced in Section 2.7. We used the implementations of Apriori, Eclat, FP-growth by Borgelt [63] as they are the state-of-art implementations of these algorithms. FP-growth* is an optimized version of FP-growth that applies an array technique to reduce FP-tree traversal time. FP-array is an improvement of FP-growth* that implements the data structure of FP-tree in the form of

arrays. AIM2 is an improvement of Eclat [61] and a combination of many optimization techniques.

Table 2-6: Descriptions of the experimential datasets

Datasets	Descriptions
Chess	Chess Endgame data
Connect	All legal positions in the game of connect-4
Mushroom	Samples of 23 species of gilled mushrooms data
Pumsb	Census population and housing data
Accidents	Traffic accident data
Retail	Retail market basket data
Kosarak	On-line click-stream data
Webdocs	Web document data

Hardware: The eight algorithms were tested on an Altus 1702 machine with dual AMD Opteron 2427 processor, 2.2GHz, 24GB memory and 160 GB hard drive. The operating system is CentOS 5.3, a Linux-based distribution. We use g++ for compilation.

2.8.2 Execution Time Comparison

The execution time of eight algorithms on eight datasets with various *minsup* are presented in Figure 2-14. The experimental results show that FEM and DFEM run stably and outperform the others in almost all cases, while the other algorithms behave differently for different datasets. Apriori runs slowest on eight datasets but it does better than FP-growth* and FP-array for two dense datasets, Chess and Mushroom. For Retail the sparse dataset, Apriori has longer execution time compared to FEM, DFEM and FP-growth but runs faster than the others (Eclat, AIM2 and FP-array). Eclat performs better than all except AIM2, FEM and DFEM on the dense datasets. However, for sparse datasets such as Retail and Kosarak, Eclat runs slower than all except Apriori. Compared to Eclat, three algorithms FP-growth, FP-growth* and FP-array run faster for the dense

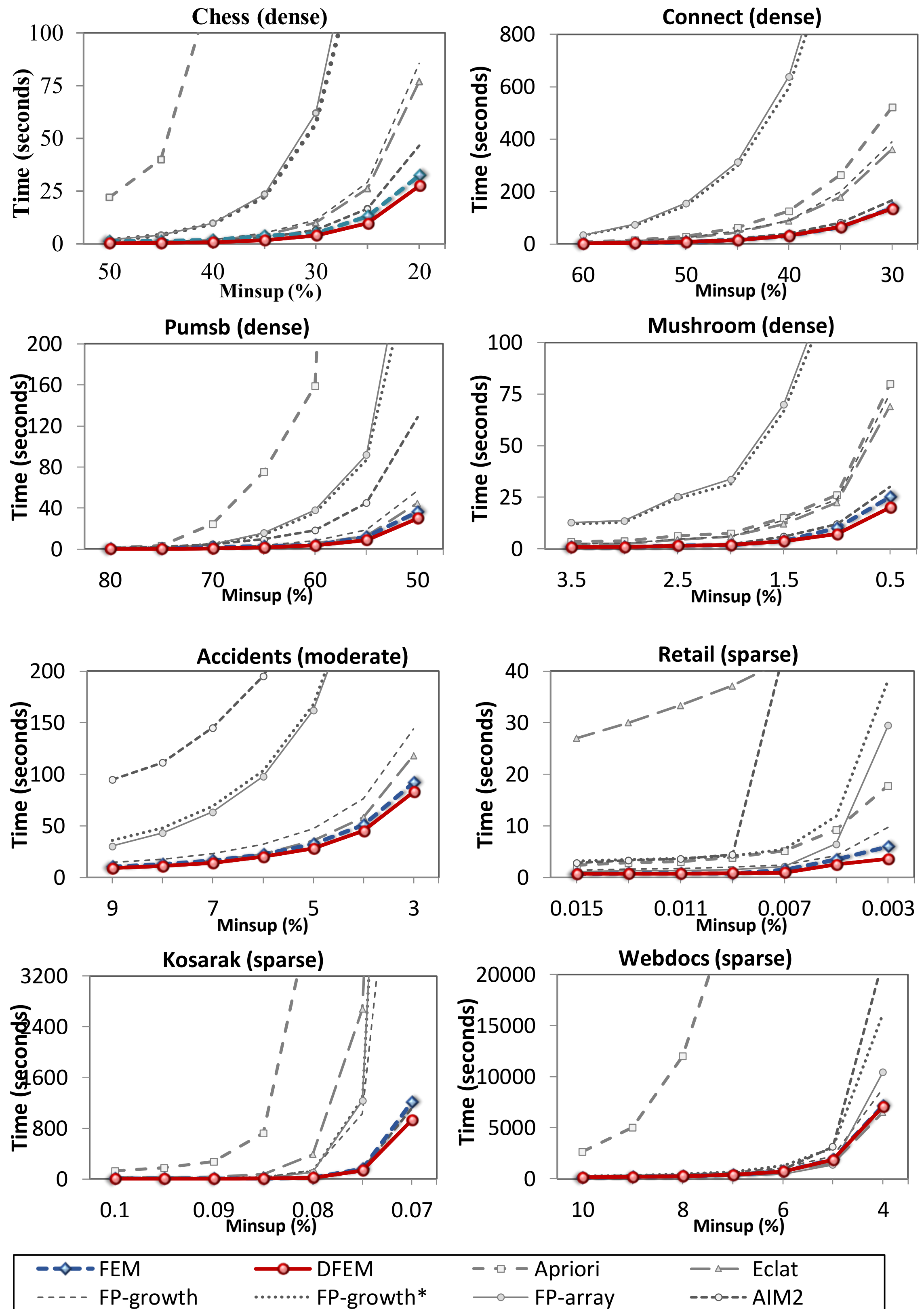


Figure 2-14: Time comparison of FEM and DFEM with other algorithms

datasets but slower for the sparse ones. AIM2, a variant of Eclat, performs well for some dense and sparse datasets but worse for others.

Based on the execution times in this experiment, we found that FEM and DFEM run faster than Apriori - most commonly used ARM method- from 3.4 to 202.43 times. In comparison to Eclat and AIM2 whose mining approach use vertical data format, our algorithms run from 1.02 to 45.3 times faster. Our algorithms performed 1.2 to 23.4 times better than FP-growth, FP-growth* and FP-array which are among the best methods for ARM. This experiment demonstrates the efficient performance of FEM and DFEM for both sparse and dense data.

2.8.3 Memory Usage Comparison

In order to evaluate the memory usage of FEM and DFEM, we measure their peak memory usage in comparison to the other six algorithms for the eight datasets by using the memusage command of Linux. Table 2-7 shows the memory usage of all algorithms for the test cases with low minimum supports so that the result can better reflect the large difference in memory usage among different algorithms. As in Table 2-7, FEM and DFEM, (in bold), consume much less memory than Apriori in every case. Their memory requirements are closer to the average memory usage of Eclat and FP-growth in most cases. For the Accidents and Connect dataset, our algorithms use less memory than both Eclat and FP-growth. For Chess dataset, FEM and DFEM need slightly more memory because our implementation includes some additional buffers to enhance performance. However, these buffers have fixed size and do not require much memory. Compared to FP-growth*, FEM and DFEM require more memory for the dense datasets but less memory for the sparse ones. In contrast, compared with FP-array, the memory usage of

FEM and DFEM is smaller for the dense datasets but larger for the sparse cases. The memory usage of AIM2 is smallest in most cases. However, the memory usage of AIM2 for Webdocs, where memory optimization is critical due to its large memory requirements, is a significantly lager than the others.

To sum up, the two experiments show that FEM and DFEM not only significantly improve the mining performance and outperform the existing "efficient" algorithms for both sparse and dense datasets they also compare well in memory requirements. Their memory consumption is much less than Apriori and FP-growth and is on average on par with the other algorithms. These results demonstrate the efficiency and efficacy of our algorithms. DFEM performs better than FEM, especially when *minsup* is low. Therefore, for mining applications that requires low *minsup* [33], DFEM is a better choice.

Table 2-7: Peak memory usage (megabytes) of FEM, DFEM and other algorithms

Datasets	**Minsup**	**FEM**	**DFEM**	**Apriori**	**Eclat**	**FP-Growth**	**FP-Growth***	**FP-array**	**AIM2**
Chess	20%	**4**	**4**	1139	2	3	3	33	1
Connect	30%	**11**	**11**	31	13	16	2	43	3
Mushroom	0.5%	**4**	**4**	20	3	5	2	33	1
Pumsb	50%	**15**	**15**	921	15	15	6	46	10
Accidents	3%	**181**	**181**	368	232	305	198	154	40
Retail	0.003%	**30**	**30**	1203	25	33	350	59	32
Kosarak	0.07%	**141**	**141**	16406	138	154	160	133	130
Webdocs	4%	**4707**	**4707**	24576	3996	5103	5581	4256	7544

2.8.4 Impact of Applying Two Mining Strategies

To study the performance merit of our mining approach, we measured the mining time of our approach in three separated cases. The first case (Case 1), only *MineFPTree* is applied to generate the complete set of frequent patterns while *MineBitVector* is disabled (setting *K=0* in FEM and DFEM). The second case (Case 2), only *MineBitVector* is used. The third case (Case 3) use both strategies by dynamically switch between *MineFPTree* and *MineBitVector* (our proposed approach). The results for DFEM on both dense and sparse data (Figure 2-15) show that it performed better than the cases where single mining strategy was used. Because the results for FEM were close to that of DFEM, only results for DFEM are presented here. For example, by applying both *MineFPTree* and *MineBitVector* DFEM runs 2.1 – 8.9 times faster for Chess dataset and 4.7 - 6.4 times faster for Kosarak when compared with the cases of using single mining strategy (Figure 2-15). This is explained by the ability of DFEM to select the suitable strategy (i.e. either *MineFPTree* or *MineBitVector*) for each subset of data being mined based on their characteristics. In Case 1, *MineFPTree* is applied to mine all data subsets although it is only suitable for sparse datasets. Performance loss occurs when *MineFPTree* mines the dense data subsets. In Case 2, all data subsets are mined using *MineBitVector* which is more suitable for dense data subsets. As a result, the performance loss occurs when it mines the sparse data portions. DFEM applies both mining strategies; it selects *MineFPTree* for the sparse and *MineBitVector* for the dense portions to improve the overall performance. It is important to note that data characteristics of mining data vary as *minsup* changes. For both Chess and Kosarak, Case 1 runs faster than Case 2 for the larger *minsup* values but slower for smaller *minsup*. This is because when *minsup*

value is reduced, more frequent patterns are generated and the number of small dense data subsets to be processed will be larger than the number of sparse data subsets making *MineBitVector* a more suitable option. In such a case, using *MineFPTree* only (Case1) not only result in a large performance loss but also it will perform worse than using *MineBitVector* only (Case 2). DFEM can detect the change of data characteristics to balance the use of its two mining strategies and hence run faster and stably for various *minsup* values.

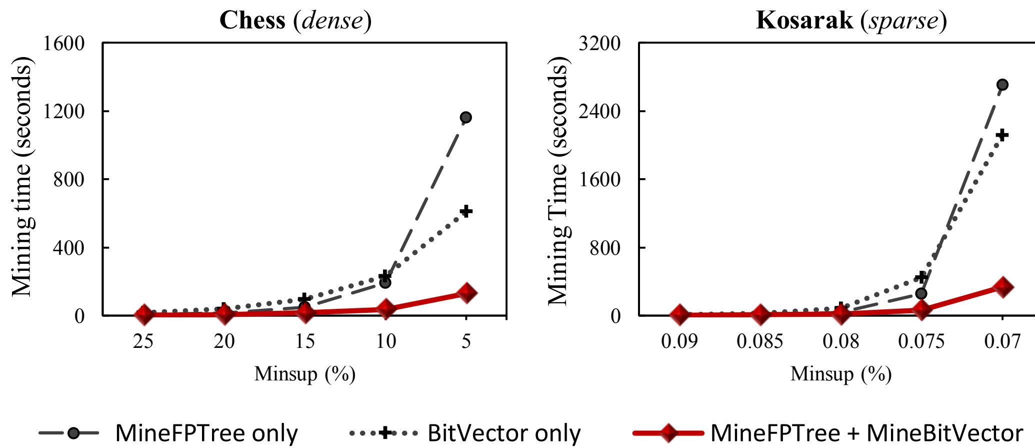


Figure 2-15: Running time of using single mining strategy vs. using both

To provide insight into the performance merits of FEM and DFEM, running times of *MineFPTree* and *MineBitVector* are measured separately to observe the contribution of each mining task to the final performance. Figure 2-16 presents the result of this experiment for three datasets Chess (dense), Accident (moderate) and Webdocs (sparse). Because the time of FEM and DFEM are not much different for these datasets, we used one set of charts to present the running time distribution. The results show that *MineBitVector* is responsible for 93% - 99% of the mining time for the dense dataset Chess. This is because the shape of the FP-tree of dense datasets is usually compact and

most conditional pattern bases satisfy the condition to switch from *MineFPTree* to *MineBitVector*. In contrast, for the very large and sparse dataset Webdocs, *MineFPTree* is responsible for 90% - 99% of the mining time because many large FP-trees are generated and most of them do not satisfy the switching condition. For the Accidents dataset whose density is moderate, the running time distribution of FEM and DFEM is balanced between the two mining strategies. We found that this running time distribution pattern of dense, moderate and sparse datasets is consistent for the other datasets as well. It must be noted that the running time distribution does not indicate the amount of work. In fact, *MineBitVector*, by using faster bitwise operations and more cache-friendly data layout, will process larger amounts of data than FP-Tree mining does in the same unit of time.

The running time distribution changes when minimum support varies. When the minimum support is set to lower levels, the FP-tree constructed from the original dataset is larger with more branches of new frequent items added to the bottom of the tree. These branches are usually small if the minimum support is very low. As a result, more small conditional FP-trees are generated which satisfy the condition to switch from *MineFPTree* to *MineBitVector*. This explains why the running time percentage of *MineBitVector* increases as the minimum support is reduced (Figure 2-16). In conclusion, FEM and DFEM have the ability to switch between strategies at runtime by distributing the mining workload to the appropriate mining strategy that fit the characteristics of the database being processed.

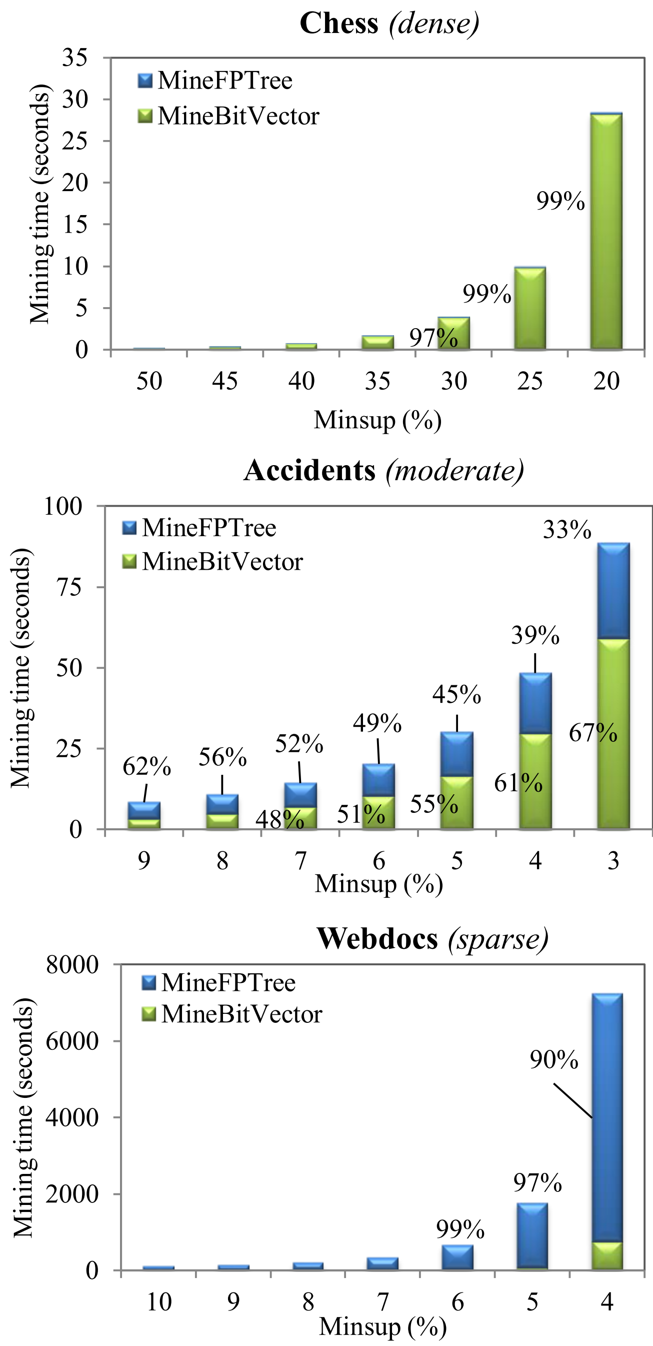


Figure 2-16: Running time distribution of two mining strategies in DFEM

2.9 Conclusion

FEM and DFEM are two novel FPM algorithms that run fast and save memory on both sparse and dense databases by applying our data characteristics based approach. We have introduced a combination of several optimization techniques for FEM and DFEM to further enhance their performance. The experimental results show that FEM and DFEM significantly improve FPM performance and outperform previous developed efficient algorithms on both sparse and dense databases. In addition, they consume much less memory than Apriori and FP-growth. Of the two, DFEM runs faster in most test cases. The proposed methods presented in this chapter are not limited to efficient sequential FPM, it is important that they can be used as the base mining strategies for our parallel FPM methods presented in Chapters 2, 3 and 5.

3. Parallel Frequent Pattern Mining on Shared Memory Multi-core Systems

3.1 Introduction

In Chapter 2, two new effective and efficient sequential FPM methods, FEM and DFEM, were described and analyzed. Although these methods provide better performance than the compared sequential methods [8], [25], [26], [29], [31], [32], their sequential execution time is large when mining large databases or with small *minsup* input values due to the computational intensive nature of FPM. Parallel computing provides much needed computing power for which new FPM algorithms can be designed to speed up the FPM mining computation. We present in this chapter our parallel FPM solution for machines that have multiple cores where cores share single memory space (i.e. shared memory MIMD).

3.1.1 Motivation

For a data intensive problem like FPM, efficient use of memory can dramatically speed up the execution time [29]. However, most existing parallel FPM algorithms under-utilize the benefits of shared memory as they are mostly designed to use only the distributed memory programming model [33], [34], [35], [36], [37], [38]. This is usually attributed to the ability to run applications developed for a distributed memory environment on shared memory platform. The opposite is not easily accomplished. Recent surveys [34], [35] show that very few studies are conducted on parallel FPM algorithms for shared memory multi-core computers although they constitute the compute node of high-end HPC machines. None of existing parallel FPM work take into account the data characteristics to improve the performance for different database types and based on the data characteristics of datasets being mined.

3.1.2 Contributions

To address these issues, we develop ShaFEM, a novel parallel method for mining frequent patterns that performs efficiently on multi-core shared memory systems. In ShaFEM, we employ the mining approach of DFEM (Chapter 2) and redesign the way that data are processed so that we can best utilize the available shared memory to obtain high performance of FPM on sparse and dense databases. We develop a new data structure, XFP-tree, to enhance the locality of data processed by multiple cores in the system. The main contributions of this work are:

1. ShaFEM is a new efficient parallel lock free approach that applies newly developed data structures to enhance independence of parallel processes, minimize synchronization and improve cache utilization (Section 3.4). Its dynamic job scheduling for load balancing enhances CPU utilization and scalability of FPM on multi-core shared memory systems. This is an important contribution because FPM, which has many dependent subtasks, unpredictable workload and complex data structures, is a challenging HPC problem.
2. A new XFP-tree data structure designed to enhance parallelism provides independence between parallel processes where they work on their sub datasets (Section 3.5); independently and dynamically switch between two mining strategies (one suitable for sparse and the other suitable for dense datasets), allowing for more accurate and optimal switch decisions (Section 3.6).
3. We demonstrate the efficiency of ShaFEM by conducting intensive experiments to benchmark ShaFEM and other state-of-the art FPM methods. An in-depth analysis of ShaFEM performance characteristics is conducted and presented (Section 3.7).

3.2 Background

3.2.1 Architecture of Multi-core Shared Memory System

Multi-core architectures are equipped with multi-core processors where each processor (also referred to as chip or socket) consists of two or more cores. The number of cores typically varies from 2 to 8 in low-end machines (e.g. mobile devices, PCs, etc.) and from 4 to 64 in high-end computers (e.g. multi-core multi-socket servers) where the system memory is shared. Each core in a processor can independently execute a process or thread belonging to a parallel program (or independent jobs). All cores of the systems access the same memory space. Large servers usually include two or more sockets connected via very high speed interconnection network. The memory access methods in these systems may be either UMA (Uniform Memory Access) or NUMA (Non-Uniform Memory Access) [64]. Figure 3-1 demonstrates a 12-core shared memory system equipped with dual 6-core sockets; cores of a socket have private L1/ L2 cache and share L3 cache. The twelve cores share the system memory.

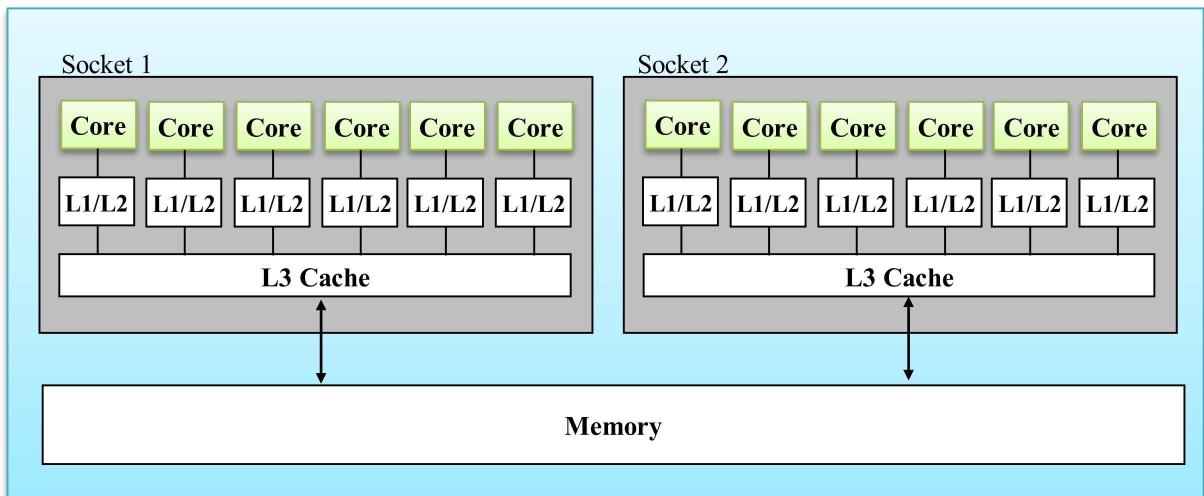


Figure 3-1: Architecture of multi-core shared memory system

3.2.2 Parallel Programming Models for Multi-core Shared Memory Systems

In order to leverage the computing power of multi-core shared memory systems, the parallel programs must have the ability of creating processes, partitioning and assigning the workload to processes so that they can cooperatively execute on multiple cores. The two most commonly used programming models in development of parallel programs for these systems are shared-memory programming and distributed-memory programming models.

Shared memory programming model- parallel programs of this model create a group of threads/processes to execute concurrently the parallel code regions on many cores/processes. Their threads/processes access same memory address space. This model has two advantages. First, it enables low latency and transparent data communication between parallel execution units (i.e. threads or processes) using shared memory. Second, use of shared memory provides an easy and efficient way to implement dynamic job scheduling and load balancing, the two critical problems in developing parallel FPM methods. A common problem of parallel FPM programs is synchronization of concurrent memory accesses, which is costly and is needed to ensure data integrity. In our research, we propose to use OpenMP [65] & C++, a widely used shared memory parallel language.

Distributed memory programming model- parallel programs of this model create a group of processes with private memory address space. The data communication among them are therefore, done via messages passing which has higher latency, compared to shared memory. Since message passing is the only means for job scheduling and load

balancing in these programs, they do not utilize the system shared memory. We discuss ~~about~~ this programming model in more detail in Section 4.2.2 of Chapter 4.

3.2.3 FPM Challenges on Multi-core Shared Memory Systems

FPM has irregular and unpredictable workload which poses major challenges when one considers partitioning the workload for assigning and distributing to parallel processes for balanced load and optimal performance. Due to data dependence in multiple reduction steps, large synchronization cost may be incurred which limits the scalability as the number of cores used to run FPM increases. Since memory is intensively used in FPM, it is essential to design data structures that allow parallel processes to work independently to maximize the parallelism. However, these data structures usually require more memory than those of the sequential FPM methods, resulting in more cache miss and hence performance reduction. The trade-offs between memory usage and synchronization are often made to enhance performance. When many processes have to write data to the same memory address at the same time, they use synchronization, typically implemented by ”lock”, a special type of data-oriented synchronization in parallel programs, to control processes to access the shared memory elements one at a time. Heavy use of synchronizations, however, can slow down the execution as processes compete for the lock.

3.3 Related Literature Review

Very few studies proposed parallel FPM methods for multi-core shared memory machines [34], [35]. Among those are the ones inspired by FP-growth [33] because the divide-and-conquer approach of FP-growth naturally lends itself to parallelism [37], [66], [67]. In the traditional FP-growth-based parallel approach, parallel processes

cooperatively build a shared global FP-tree, resulting in extensive use of costly synchronization locks to access each node of the tree [35]. We present three parallel FP-growth based FPM methods for shared memory systems.

3.3.1 Tree Projection Partition Algorithm

Tree Projection method by Chen et al. partitions the FP-tree into subsections with small portions shared among processes. Only access to the small shared sections would require synchronization locks [68]. Although this approach reduces the synchronization cost considerably, it adds the overhead of extra partitioning of the workload and is harder to load balance. Moreover, updating of the shared portion constitutes a considerable workload of the FP-tree construction which can reduce scalability of the algorithm as the number of processes increases.

3.3.2 MLPT Algorithm

Multiple Local Parallel Trees (MLPT) by Zaiane et al. is the first algorithm not requiring locks by constructing local FP-trees separately and mining the frequent patterns from these trees [69]. This approach has shown good scalability on shared-memory multi-core machine.

3.3.3 FP-array Algorithm

The parallel version of FP-array by Liu et al. [29] by is another efficient algorithm that uses locks for FP-tree construction. It then converted this data structure into arrays for better cache optimization. This method significantly improves performance compared to the previous parallel methods and has been integrated into the PARSEC Benchmark [70]. Frequent patterns are generated by recursive construction of child FP-trees from the parent FP-tree. Because of the divide-and-conquer approach of

FP-growth, the mining workload can be partitioned and distributed to parallel processes without data dependence conflicts.

Due to inheriting the mining characteristic of FP-growth, the above parallel methods have poor performance on dense databases. In our study, we focus on solving this issue and propose a parallel solution that works efficiently on shared memory multi-core machine architecture.

3.4 ShaFEM Algorithm

3.4.1 Overview

ShaFEM, our new parallel FPM for multi-core share memory systems, performs its mining task in the following two stages:

- *XFP-tree construction stage*- ShaFEM applies the FP-tree based approach to compact all data in memory to avoid high cost of I/O due to multiple database scans. The database is divided into equal parts; each parallel process reads its portion of data to construct its local FP-tree. The local FP-trees are then merged into a global XFP-tree that is shared among processes. The trees are implemented and constructed without the need for locks for minimizing synchronization cost and enhancing scalability.
- *Frequent pattern generation stage*- all frequent patterns are found using divide-and-conquer approach. The frequent items in the header table of the XFP-tree are dynamically obtained by the parallel processes as they become available in order to balance the workload. This self-scheduling helps to balance the workload among the processes. Each parallel process recursively and independently generates all frequent patterns ending with one item being assigned and continues

with the next item. Similar to DFEM, ShaFEM uses two mining strategies for frequent pattern generation: FP-tree that uses a horizontal data format, and bit vector that uses a vertical data format. A process will dynamically switch between the two strategies in the course of mining for frequent patterns depending on the density of the remaining data to be mined.

3.4.2 Data Structures

FP-tree is a prefix tree storing all sets of ordered frequent items as described in Chapter 2.

XFP-tree is an extension of FP-tree newly introduced in ShaFEM. This data structure stores all sets of frequent items retrieved from the database, and differs from FP-tree because some degree of node duplication is allowed. It is constructed by combining several FP-trees into a single tree described in detail in Section 3.5. An XFP-tree (Figure 3-4) is purposely designed so that it is not as compact as a FP-tree (Figure 2-2) in order to achieve higher degree of parallelism and scalability. This data structure is customized for parallel access and does not require synchronization during concurrent construction as processes can create replicated nodes instead of updating the same node.

Bit Vector presents occurrence of a set of items in databases in vertical format. The data structures of bit vector and its supplement weight vector are described in Chapter 2.

3.5 Parallel XFP-Tree Construction

In the first stage of ShaFEM, the global XFP-tree, shared among all cores, is built. This process involves three main steps:

Step 1 - Finding the frequent items:

1. The database is evenly divided into horizontal partitions and is distributed to parallel processes. For example, assuming three processes, the dataset in Table 2-2 is partitioned into 3 parts (Figure 3-2a)

2. Each process reads its data partition and computes a local count list of all items in the databases (Figure 3-2b). Data is read in parallel by all processes without synchronization because the database is equally partitioned and each process can determine its data partition using its process ID. Data are read and processed in blocks to reduce I/O overhead.

3. A parallel summation is performed to reduce the local *count* lists into a shared global *count* list. Each process P_i is responsible for a number of elements in the global *count* list to compute their count (Figure 3-2c). We do not need to implement locks for reduction because each process works on separate set of items.

4. The frequent items are identified and sorted in the descending order using their count and the user-supplied *minsup* (Figure 3-2c). In the example, given *minsup* = 20%, a, b, c, d, and e are frequent items and f is infrequent because its *count* is 1 and its *support* is 11.1% which is smaller than the specified *minsup*.

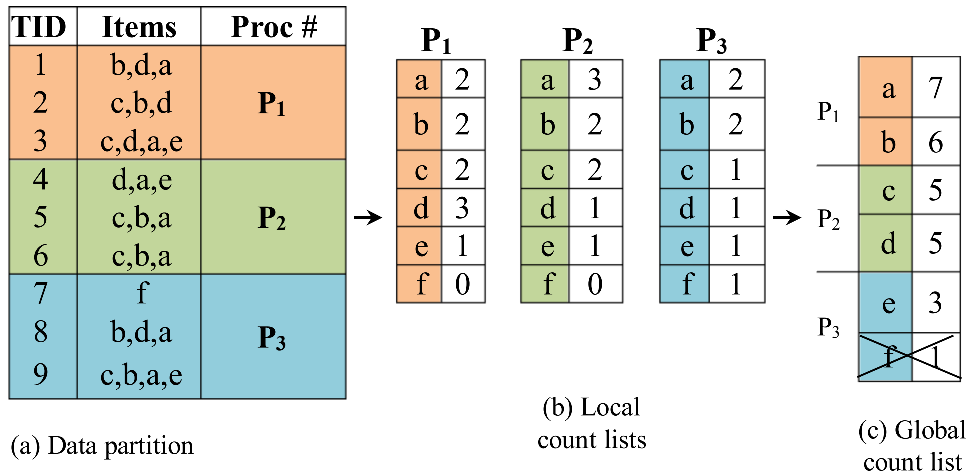


Figure 3-2: Parallel construction of the global count list

Step 2 - Constructing the local FP-trees:

1. Each process creates a local header table consisting of the sorted frequent items and their local *counts*.

2. Each process reads the transactions from its data portion for the second time to get frequent items of each transaction and inserts them into an FP-tree in their frequency descending order. This is the most time consuming step of the first stage and in our design, all processes work independently to build their local FP-trees. Figure 3-3 presents the three local FP-trees created concurrently from the same dataset.

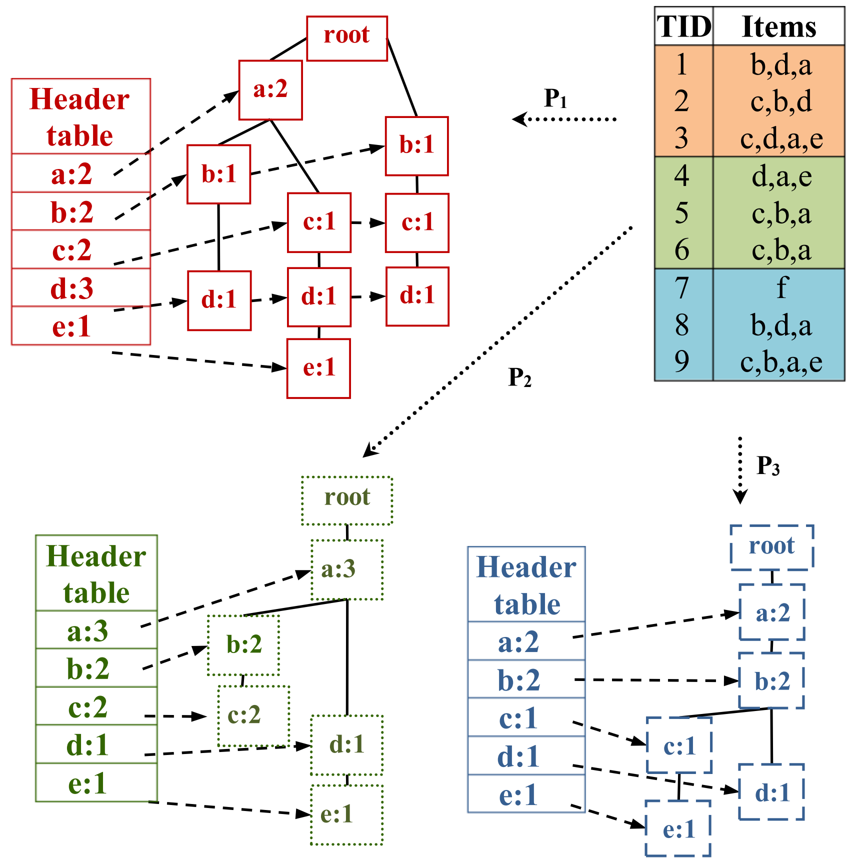


Figure 3-3: Local FP-tree construction

Step 3 - Merging local FP-trees into a global XFP-tree:

1. The construction of the global XFP-tree is initialized by converting the header table of one local FP-tree into the header table of the global XFP-tree. The frequent items in this table are divided into even subsets and assigned to the parallel processes. For example, *a, b* are assigned for P_1; *c, d* for P_2 and *e* for P_3. Each P_i updates items of this table with the global *count* using the global *count* list of Step 1.

2. Each process P_i then joins the local linked lists of their assigned items in the local FP-trees into the global ones by starting from the existing linked list of the global

header table. When all processes complete their work, the XFP-tree is created as in Figure 3-4. The time to perform this step is negligible because the manipulation of linked lists can be performed in parallel without changing the local FP-trees. Because the next pattern mining stage uses this XFP-tree by traveling in bottom-up direction, the root node of XFP-tree is not needed and will not be created.

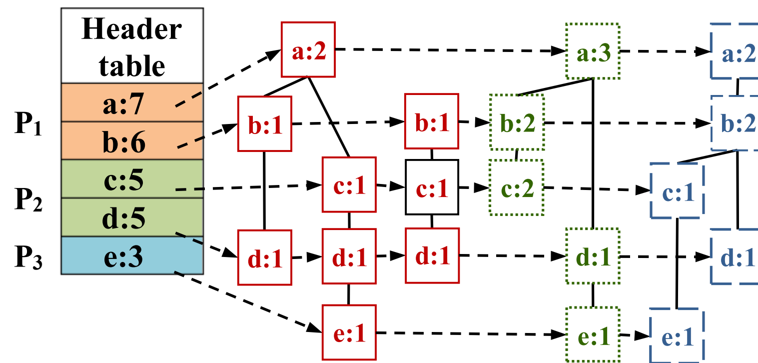


Figure 3-4: The global shared XFP-tree

3.6 Parallel Frequent Pattern Generation

3.6.1 Parallel Frequent Pattern Generation Based on Data Characteristics

The second stage is similar to the multiple strategies introduced in DFEM. ShaFEM generates all frequent patterns by exploring a very large number of data subsets extracted from the database and applying one of the two mining strategies *MineFPTree* or *MineBitVector* for each data subset based on its data characteristics. This approach is distinct from prior related parallel works [35], [68], [70], [69] which applied a single mining strategy.

Figure 3-5 presents the overview of the parallel frequent pattern generation process. After global XFP-tree is constructed, parallel processes independently start searching for frequent patterns using the three tasks *ParallelMinePattern*, *MineFPTree* and *MineBitVector* as described following in more details

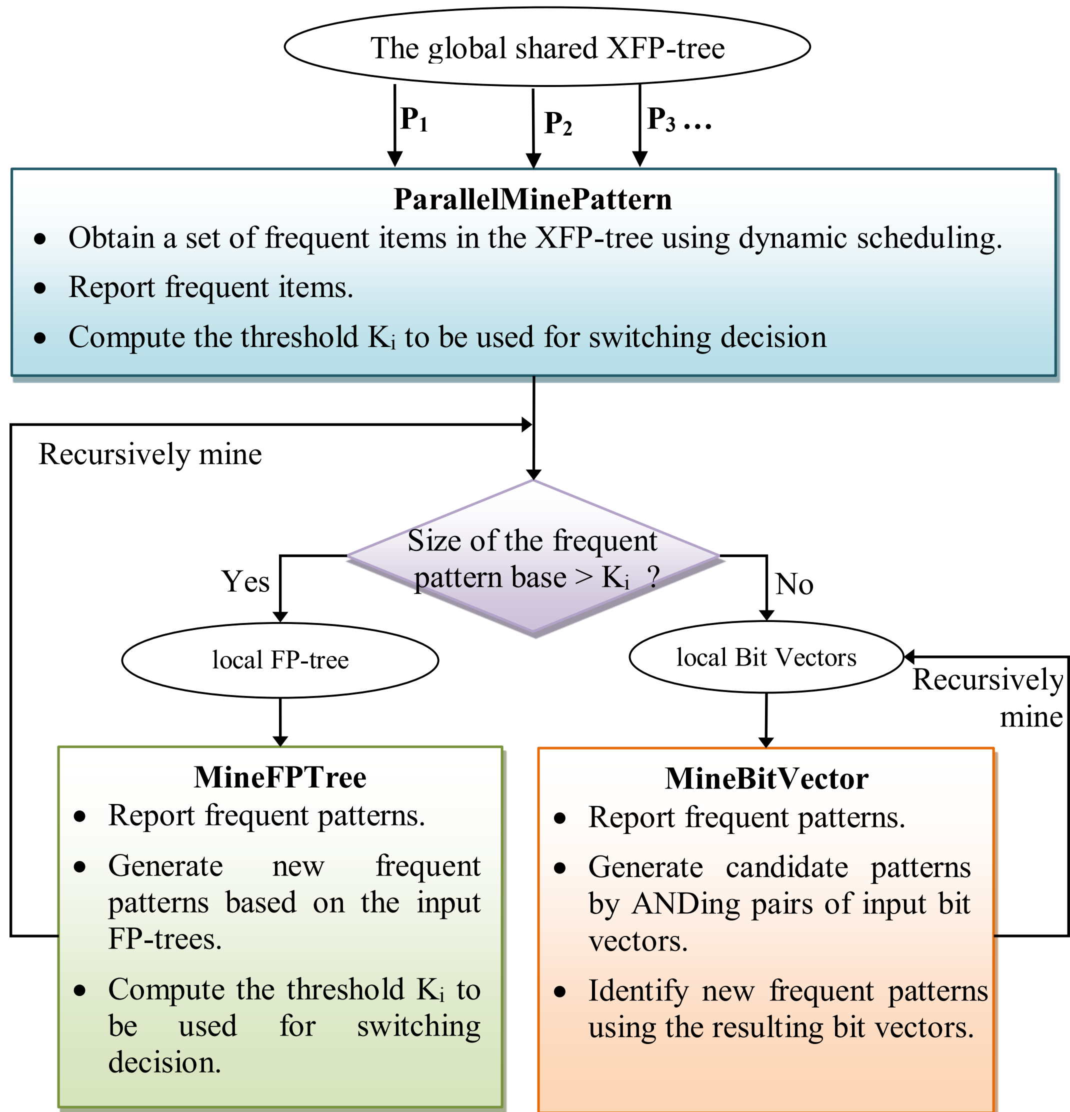


Figure 3-5: The frequent pattern generation model of each parallel process

ParallelMinePattern initializes the frequent pattern generation stage and manages the work distribution of parallel processes using dynamic job scheduling. Each parallel process P_i is assigned a frequent item α in the header table of the XFP-tree. It then traverses the XFP-tree in a bottom up direction, starting from the nodes in the linked list of the assigned item, to retrieve its conditional pattern base *C*. The dynamic decision making to switch between the two mining strategies, *MineFPTree* and *MineBitVector*, is based on the size of the conditional pattern bases in comparison with a threshold value,

K_i, which is estimated and updated at runtime using the number of frequent patterns found by the two mining strategies. If *MineFPTree* is invoked, the parallel process will build the local conditional FP-tree of item α. Otherwise, local bit vectors are generated using the base *C* and *MineBitVector* is applied. A weight vector *w* whose elements indicate the frequency of sets in base *C* is added as input to *MineBitVector;* this vector is used to compute the *count* of candidate patterns. Each parallel process P_i maintains its own threshold K_i which reflects the characteristics of the local data being processed. All parallel cores work independently until the mining process is complete. The ParallelMinePattern1 algorithm is presented in Figure 3-6.

ParallelMinePattern1 algorithm
Input: shared *XFP-tree XT, minsup* *Output:* frequent items 1: $K_i = 0$ 2: **Parallel Self-Scheduled For** *j*= 1 **to** number of items in *XT* 3: { $\alpha = j^{th}$ item in *XT* 4: Output α 5: *Size* = the size of α's conditional pattern base 6: Compute and update threshold K_i 7: **If** $Size>K_i$ **Then** 8: Construct α's local conditional FP-tree *T* 9: Call **MineFPTree3**(*T*, α,*minsup*) 10: **Else** 11: Construct α's local bit vectors *V* and *w* 12: Call **MineBitVector1**(*V*,*w*,α,*minsup*) 13: **End if** 14: }

Figure 3-6: ParallelMinePattern1 algorithm

MineFPTree generates frequent patterns using MineFPTree3 algorithm and is a parallel version of MineFPTree2 algorithm of DFEM. MineFPTree3 of ShaFEM executes concurrently on many cores of the machine to mine the data portions distributed by *ParallelMinePattern* and compute its corresponding private threshold K_i values at runtime. This mining strategy concatenates the suffix pattern of the previous steps with each item α in the header table of the input FP-tree and reports them as frequent patterns. It then constructs the conditional FP-tree of each item in the input FP-tree and recursively mines new frequent patterns from the new tree. Figure 3-7 shows the algorithmic details of MineFPTree3. The value of K_i is updated using the method described in Figure 3-8.

MineFPTree3 Algorithm
Input: FP-tree *T, suffix, minsup*
Output: set of frequent patterns
1: **If** *T* contains a single path *P* **then**
2: **For each** combination *x* of the items in *P*
3: Output $\beta = x \cup suffix$
4: Compute and update threshold K_i
5: **Else Foreach** item α in the header table of FP-tree *T*
6: Output $\beta = \alpha \cup suffix$
7: *Size* = the size of α's conditional pattern base
8: Compute and update threshold K_i
9: **If** $Size > K_i$ **Then**
10: { Construct α's conditional FP-tree *T'*
11: Call **MineFPTree3**(*T',β,minsup*) }
12: **Else**
13: { Construct α's local bit vectors *V* and *w*
14: Call **MineBitVector1**(*V,w,β,minsup*) }
15: **Endif**

Figure 3-7: MineFPTree3 algorithm

MineBitVector- ShaFEM reuses MineBitVector1 algorithm presented in Figure 2-9 of Chapter 2.

3.6.2 Switching Between Two Mining Strategies

Similar to DFEM, ShaFEM switches between the two mining strategies based on a heuristic that uses a threshold K. Each parallel process P_i maintains its own K_i and computes it by applying UpdateK2 (Figure 3-8) on the locally processed data, making more parallelism and accurate estimation of K_i because the local data characteristics vary among processes.

UpdateK2 Algorithm
Input: NewPatterns and Size *Output*: Updated values of K_i (*Initialization for the first call to UpdateK2 for process P_i: Create a private array X with N elements, Set all $X[j]$ to zero *) 1: **For** $j = 0$ to $N - 1$ 2: **If** $Size > j*Step$ **then** $X[j] = X[j] + NewPatterns$ 3: **Else** Exit Loop 4: $K_i = 0$ 5: **For** $j = N\text{-}1$ to 1 6: **If** $X[j\text{-}1] \geq 2*X[j]$ **then** $K_i = (j+1)*Step$ and Exit Loop

Figure 3-8: UpdateK2 algorithm

3.7 Performance Evaluation

In this section, we evaluate the performance of ShaFEM and compare it with prior related work.

3.7.1 Experimental Setup

Datasets: The six datasets used in FEM and DFEM evaluation are used to evaluate ShaFEM. These are real datasets with various characteristics and domains,

which include three sparse, one moderate and two dense databases obtained from the FIMI Repository [59]. The dataset features are reported in Table 3-1.

Hardware: We evaluate ShaFEM on two 12-core shared memory dual-socket servers: one with Intel Xeon processors and the other with AMD Opteron processors. Their specifications are listed in Table 3-2. Because the experimental results on both machines are consistent, for simplification and due to our AMD cluster being dedicated to parallel processing without interference from other jobs, we mostly present the results collected from the machine with AMD processors. A performance comparison of ShaFEM on the two machines to show the performance consistency of our algorithm is provided.

Table 3-1: Experimental datasets of ShaFEM

Dataset	**Type**	**# of Items**	**Average Length**	**# of Trans.**
Chess	Dense	76	37	3196
Connect	Dense	129	43	67557
Accidents	Moderate	468	33.8	340183
Retail	Sparse	16470	10.3	88126
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Software: ShaFEM has been implemented using our computational method presented in Sections 3.4 and 3.5. Furthermore, we have applied some optimization techniques presented in Section 2.7. We study the performance of ShaFEM and compare it with FP-array [29], a state-of-the-art parallel mining method for FPM based on FP-growth. The implementation of FP-array can be found in the PARSEC Benchmark Suite,

publicly available at [70]. In addition, we have benchmarked ShaFEM, running sequentially on one core, with the sequential algorithms Apriori [8], Eclat [31], FP-growth [25] and FP-growth* [26] to compare their performance on sparse and dense databases. The implementations of these algorithms are available at [59], [63], [70]. The algorithms were implemented using C/C++. ShaFEM and FP-array are parallelized using OpenMP. We use g++ for compilation.

Table 3-2: Test machines

Name	Machine 1	Machine 2
Total cores	12	12
Num. of sockets	2	2
Cores/socket	6	6
Processor Model	AMD Opteron 2747	Intel Xeon E5-2640
Architecture	Istanbul	Sandy Bridge-EP
Clock rate	2.2Ghz	2.5Ghz
LLC/socket	6MB	15MB
Memory	24GB	128GB
OS	Cent OS 5.8 (Linux)	CentOS 6.4 (Linux)

3.7.2 Execution Time

To demonstrate the efficiency of our proposed method, we study the performance of ShaFEM and compare it with FP-array [29], one of the best parallel FPM methods for multi-core shared memory architectures. FP-array inherits the mining features of FP-growth and was shown to run sequentially much faster than many sequential mining methods including FP-growth [25], nonordfp [28], AIM2 [32], kDCI [71] and LCM2 [72] on sparse databases. Unlike ShaFEM, that constructs a new data structure named XFP-tree, FP-array constructs the FP-tree in parallel and distributes its data to parallel

processes using a tiling technique. Then, it converts the global FP-tree into arrays and mines frequent patterns from this data structure. We present the execution time of ShaFEM and FP-array for the test datasets in Figure 3-9.

ShaFEM outperforms FP-array for all test cases for different number of cores and different datasets. ShaFEM runs 2.1 -5.8 times faster than FP-array for the same number of parallel processes in every case for all datasets. It is important to note that for large datasets such as Kosarak, this speedup of 2.8 for 12 cores translates to a savings of 12.8 execution hours. Sequentially, our code runs faster by 117.3 hours or 4.9 days. Although the size of Kosarak is smaller than Webdocs, this dataset was benchmarked with very low *minsup,* and thus, its execution time was longer than Webdocs.

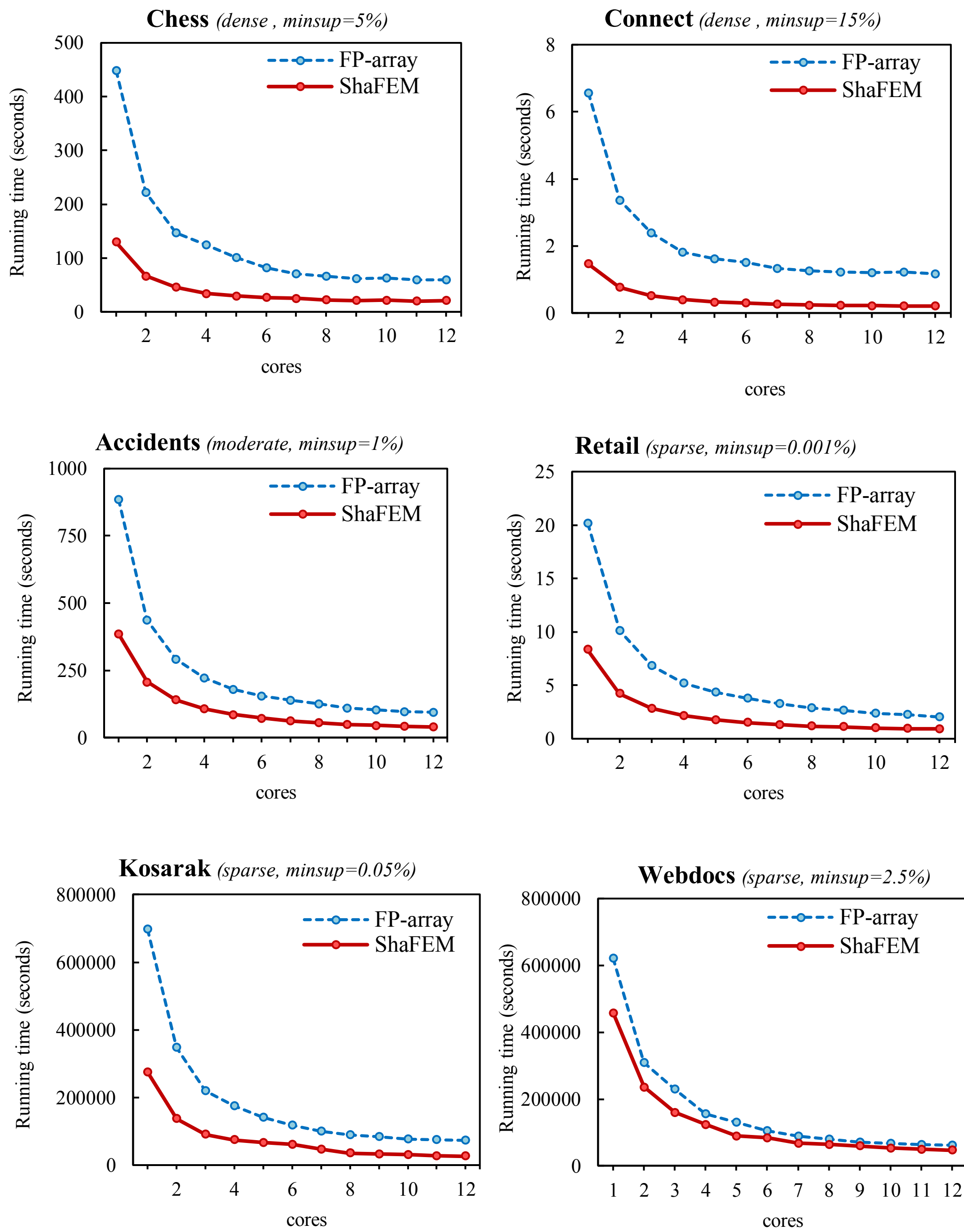


Figure 3-9: Running time comparison of ShaFEM and FP-array

Table 3-3 shows the result of running ShaFEM and FP-array on two machines with Intel and AMD processors described in Table 3-2 using 12 cores. The results show that ShaFEM performs better than FP-array for all datasets on both machines.

Table 3-3: Time comparison (sec.) of ShaFEM vs. FP-array on different hardware

Dataset	Minsup	Time on Machine 1 (AMD)			Time on Machine 2 (Intel)		
		ShaFEM	FP-array	Time difference	ShaFEM	FP-array	Time difference
Chess	5%	21	60	39	17	35	18
Connect	15%	0.2	1.2	1	0.2	0.7	0.5
Accidents	1%	41	94	53	32	53	21
Retail	0.001%	0.9	2	1.1	0.7	1.5	0.8
Kosarak	0.05%	25941	72092	46151	19719	40713	20994
Webdocs	2.5%	47631	62893	15262	32434	36274	3840

3.7.3 Speedup

Figure 3-10 shows the speedup on 12 cores of ShaFEM and FP-array compared with the sequential running time of FP-array. These results show that ShaFEM runs significantly faster than both sequential and parallel FP-array. Compared to the execution time of FP-array on one core, ShaFEM on 12 cores performed 13 – 31.3 times faster while FP-array on 12 cores has been 5.6 – 10.0 times faster than its sequential execution time.

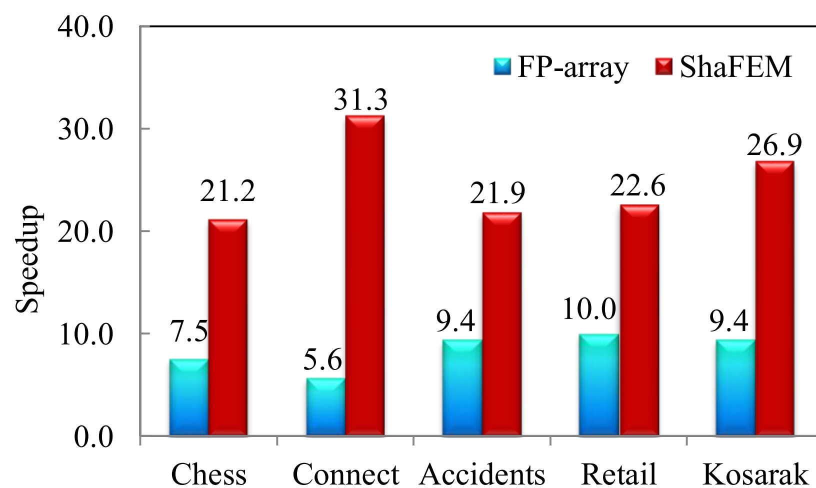


Figure 3-10: Speedup of ShaFEM and FP-array on 12 cores relative to FP-array 1 core

Figure 3-11 shows the speedup of ShaFEM for different number of cores compared to its sequential time on one core. When all 12 cores were used, ShaFEM runs 6.1 - 10.6 times faster than it does on a single core. ShaFEM scales better for sparse datasets with near linear scalability for Accidents, Retail and Kosarak. For the dense datasets Chess and Connect, speedup increases slower when 8 - 12 cores are used. This is due to the nature of the dense (compact) data structure that makes achieving good load balance for higher number of cores difficult. FP-array suffers from similar situation as can be seen from its speedup of 5.6 on 12 cores for dense dataset Connect. Both Figure 3-10 and Figure 3-11 demonstrate effectively that the ShaFEM is scalable and its performance will continue to improve with additional cores and larger datasets.

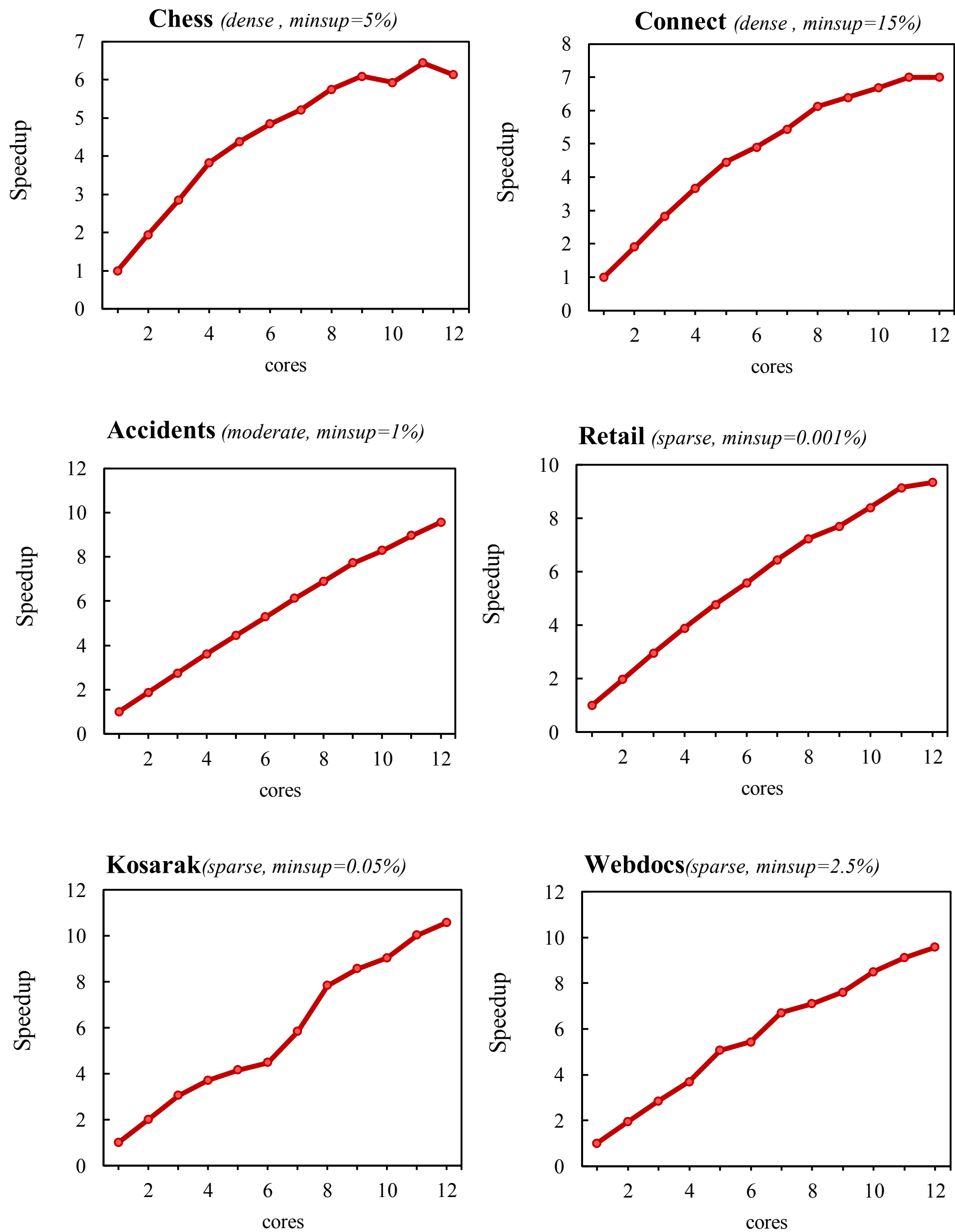


Figure 3-11: Speedup of ShaFEM on 12 cores compared to its time on 1 core

3.7.4 Memory Usage

In order to evaluate the memory usage of ShaFEM, we measure the peak memory usage in comparison to FP-array for the six datasets by using the *memusage* command of Linux. Figure 3-12 shows their memory usage for different number of cores. As the

figure shows, ShaFEM consumes much less memory than FP-array in most test cases. Specifically, the peak memory usage of ShaFEM was 1.5 – 7.1 times less than FP-array for Chess, Connect, Retails, Kosarak and Webdocs. For Webdocs dataset using 12 cores, ShaFEM used 7GB of memory less than FP-array. The only case that ShaFEM used more memory than FP-array was for Accidents dataset; their memory usage difference was 23% – 39%.

Although the memory usage of ShaFEM is 23%-39% higher than FP-array in this case, ShaFEM runs faster than FP-array. The way in which the allocated memory is accessed and processed has a high impact in the execution time. The new XFP-tree data structure allows for more lock free parallel access, less memory contention and better cache locality. The bit vector data structure allows for better memory/cache alignment. Our profiling information for Accidents using the CodeAnalyst tool [73] shows that, compared to FP-array, ShaFEM had 61% less data cache access, 59% less miss alignment access, 40% fewer branches and 21% less branch miss predictions.

The memory usage of both ShaFEM and FP-array increased as more parallel cores were employed, but the rate of memory usage increase for ShaFEM was smaller than that of FP-array. For the memory intensive problems like FPM, efficient utilization of memory has significant impact on the execution time. We use the bit vector structure to save memory and arrange data elements to increase data locality for cache optimization, and as results show for most cases use less total memory than FP-array and have faster execution time for all cases.

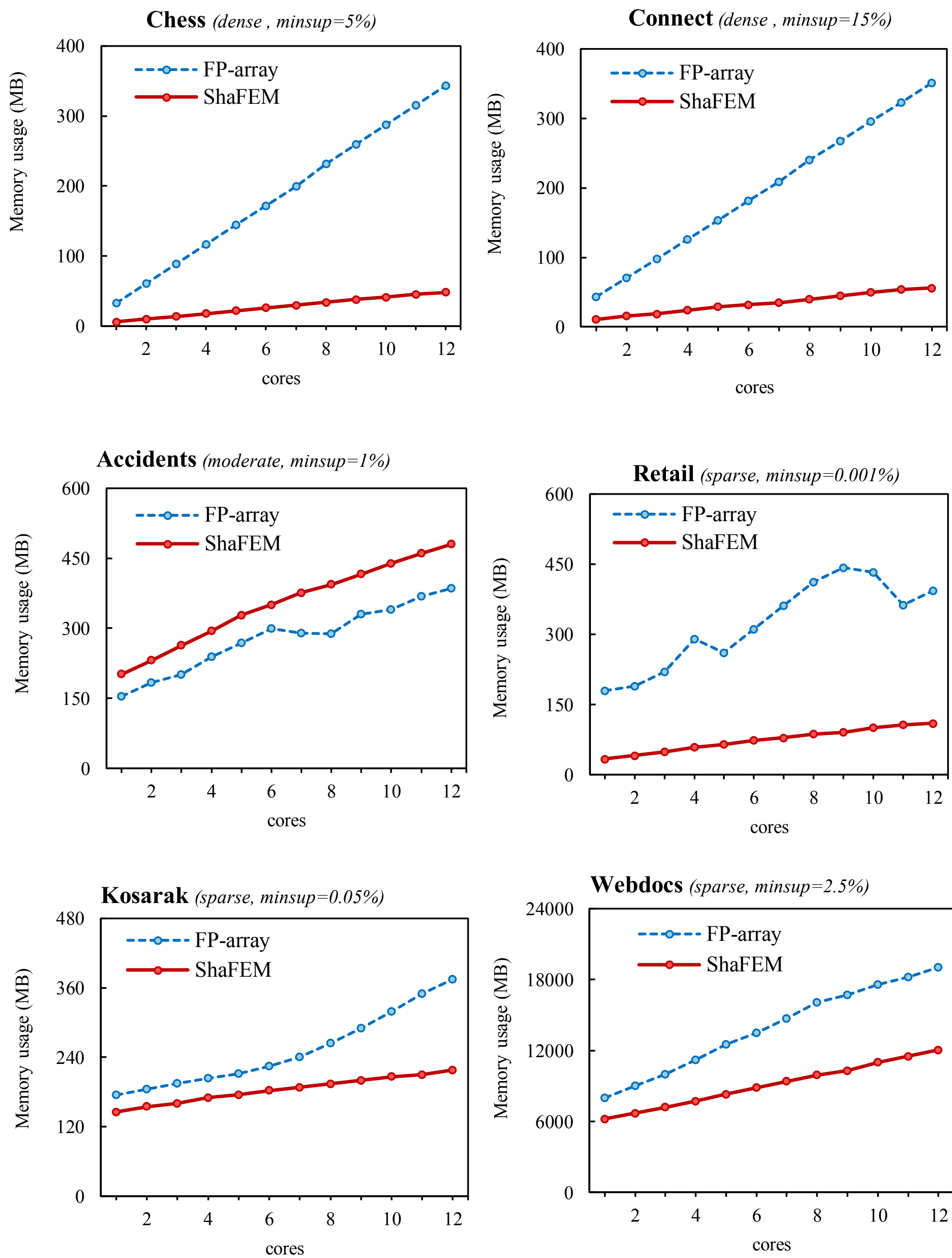
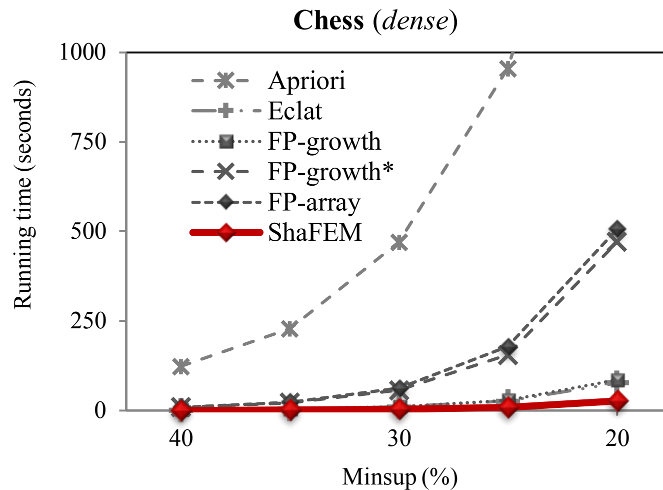


Figure 3-12: Peak memory usage (megabytes) of ShaFEM and FP-array

3.7.5 Sequential Performance Evaluation

ShaFEM also outperforms other well-known sequential FPM algorithms. This is demonstrated by benchmarking ShaFEM with Apriori [8], Eclat [31], FP-growth [25] and FP-growth* [26], FP-array [29]. The features of these algorithms are described in Section 2.3. Section 2.8 shows the performance of these algorithms, compared with that of FEM and DFEM. In this experiment, all algorithms run on a single core for test cases of various minimum support values (*minsup*). It is important to note that, when running in sequential mode, ShaFEM also uses FP-tree for finding frequent patterns, its performance is, therefore, not much different from that of DFEM and FEM.

Results show that ShaFEM outperforms the compared methods for all test cases. In addition, we also observe the performance inconsistency of the other methods for different database characteristics and minsup values. We take the performance results on Chess (dense) and Kosarak (sparse) datasets (Figure 3-13) as an example. While ShaFEM performed best for all cases, FP-growth, FP-growth* and FP-array performed better than Eclat on the sparse dataset but worse on the dense ones. Apriori performed worst for all cases.



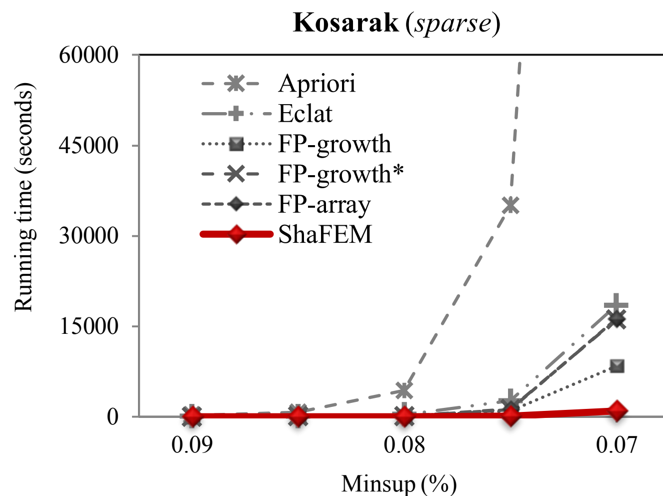


Figure 3-13: Sequential running time of ShaFEM compared to sequential methods

Figure 3-14 presents the speedup running ShaFEM sequentially compared to the test algorithms for Chess (dense) and Kosarak (sparse) datasets. ShaFEM runs much faster than the compared methods on both dense and sparse dataset for various *minsup* inputs. For example, in our test cases, ShaFEM runs 43.8 – 323.4 times faster than Apriori; 2.2 – 19.9 times faster than Eclat; 2.0 – 9.0 times faster than FPgrowth, 2.6 – 17.4 times faster than FP-growth* and 1 – 18 times faster than FP-array for the range of *minsup* values indicated in Figure 3-14.

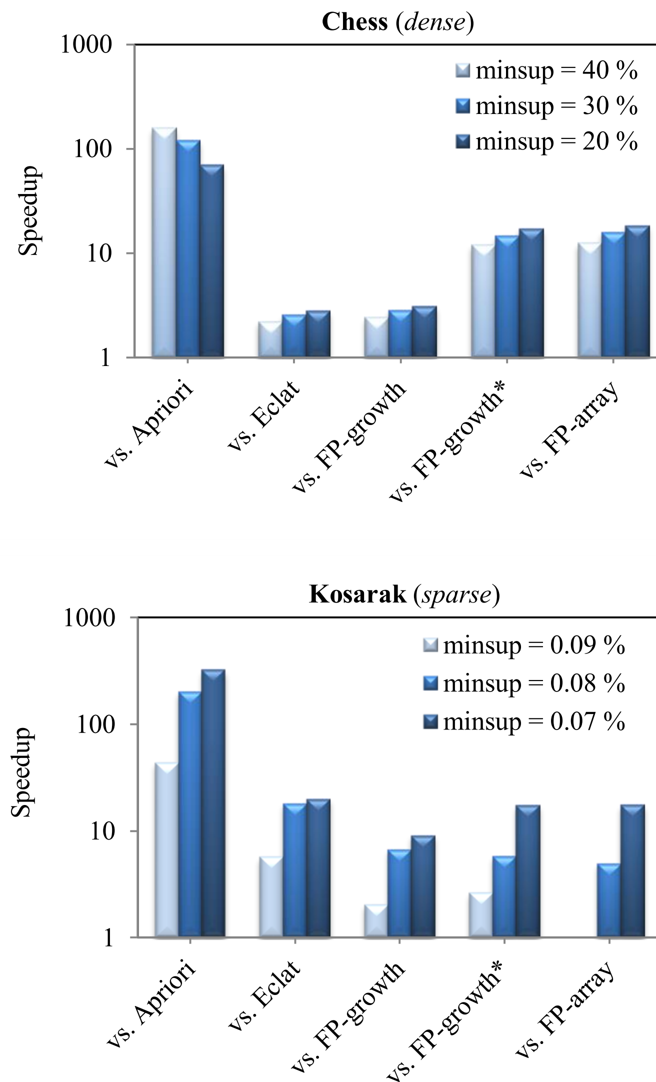


Figure 3-14: Speedup of ShaFEM on 1 core compared to sequential algorithms

3.7.6 Analyzing Performance Merits of ShaFEM

We analyze the key elements that play important roles in performance enhancement of ShaFEM including (1) the application of two mining strategies in ShaFEM to better adapt to data characteristics; (2) the dynamic scheduling method of ShaFEM to obtain better load balance; and (3) adoption of a new lock free approach in the construction of XFP-tree.

3.7.6.1 The impact of combining two mining strategies

To study the self-adaptive ability of ShaFEM to data characteristics, we measure the amount of time that ShaFEM spends on each of its two mining strategies separately when both strategies have been applied. Figure 3-15 presents the percentage of time distribution for the six test datasets. The results show that both mining strategies of ShaFEM contributed to generate the frequent patterns. However, their percentage of contributions varied depending on the data characteristics of each dataset. The mining strategy using Bit Vector was utilized mostly for the dense datasets. However, the time percentage of this strategy reduced when the data were sparse. This workload distribution for the two mining strategies is done automatically due to dynamic switching between them. The mining strategy using Bit Vector is more suitable for dense data as the low cost bitwise operations are used to generate large number of frequent patterns as are usually found from this type of data. The bit vector data structure is more cache friendly, saves memory usage and boosts the mining performance. For the sparse portions of the datasets, the number of frequent patterns is smaller. Therefore, FP-tree is a better choice because it does not require to generate very large number of infrequent candidate patterns.

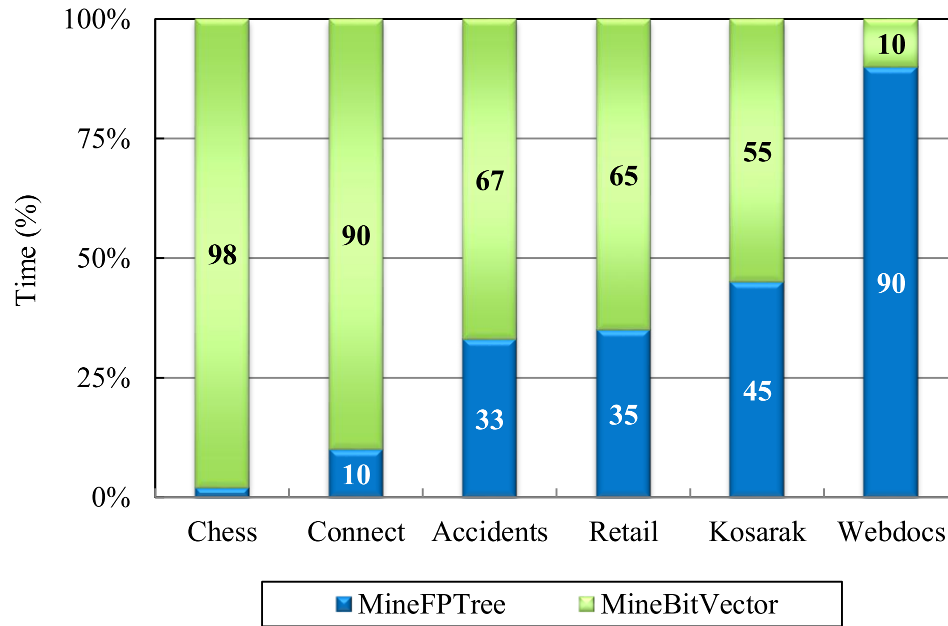


Figure 3-15: Time distribution for two mining strategies of ShaFEM

3.7.6.2 Impacts of dynamic scheduling

Generating frequent patterns, which occurs in the second stage of ShaFEM, normally accounts for much of the execution time. In this stage, load balancing is a critical issue. The divide and conquer approach is used in this stage to enhance parallelism by enabling parallel processes to work independently and select small work portions one at a time. Because the workload of each portion varies, dynamic scheduling is used to balance the load. In order to study the effectiveness of this scheduling method, we measure the execution time with two cases: (1) using static scheduling and (2) using dynamic scheduling and report the results in Table 3-4. Applying dynamic scheduling can increase the mining performance 114%-489% in our test cases.

Table 3-4: Performance of ShaFEM with dynamic vs. static scheduling on 12 cores

Databases	**Minsup**	**Static Scheduling** (1) (seconds)	**Dynamic Scheduling** (2) (seconds)	**Time Difference** (3) = (1)-(2) (seconds)	**Performance Improvement** (4) = (3)*100/ (2) (%)
Chess	5%	45	21	24	114%
Connect	15%	0.51	0.2	0.31	155%
Accidents	1%	161	41	120	293%
Retail	0.001%	5.3	0.9	4.4	489%
Kosarak	0.055%	15802	6902	8900	129%
Webdocs	3%	48183	7862	39086	497%

3.7.6.3 Impacts of lock-free approach in constructing XFP-Tree

We use a lock-free approach to construct XFP-tree, a new data structure derived from FP-tree to reduce synchronization needs and enhance parallelism. To evaluate the performance benefits of this approach, we implement a variant of ShaFEM in which parallel processes construct a single shared FP-tree (instead of XFP-tree) by using a lock for each node of the tree, where node updated are done atomically. Table 3-5 presents the execution time of ShaFEM in two cases: (1) build FP-tree with lock; (2) build XFP-tree without lock (i.e. our proposed solution). The results show that the version using XFP-tree increases the overall performance and in one case up 22.2%.

Table 3-5: Performance of ShaFEM using lock vs. lock-free on 12 cores

Databases	**Minsup**	**Build FP-tree with lock** (1) (seconds)	**Build XFP-tree without lock** (2) (seconds)	**Time Difference** (3) = (1)-(2) (seconds)	**Performance Improvement** (4) = (3)*100/ (2) (%)
Chess	5%	21.38	21.2	0.18	0.8%
Connect	15%	0.2	0.2	0	0.0%
Accidents	1%	42.69	41	1.69	4.1%
Retail	0.001%	1.14	0.9	0.24	22.2%
Kosarak	0.055%	8033	6902	1131	16.4%
Webdocs	3%	8789	7862	927	11.8%

3.8 Conclusion

We have presented ShaFEM, a novel parallel FPM method for multi-core share memory machine, evaluated and demonstrated that its performance on different database types via a number of experimental results on a 12-core machine. This dynamic parallel method that combines two mining strategies runs faster and consumes less memory than the state-of-the-art methods. It performs stably on both sparse and dense databases. ShaFEM allows the FPM task to self-adapt to the data characteristics and utilize the benefits of shared memory features in multi-core computers.

4. Parallel Frequent Pattern Mining on Multi-core Cluster

4.1 Introduction

With growth of data repositories in numerous fields, such as biology, business administration, internet, social networks, entertainment, telecommunication, etc., [1], [2], [3], [4], high performance computing (HPC) methods that use large cluster computers or supercomputers are essential. These computing platforms provide massive computing and memory resources, making them ideal for large data analysis. However, development of FPM methods for HPC requires platform-specific design to efficiently leverage platform's powerful resources.

4.1.1 Motivation

Our study of parallelizing FPM for large-scale data mining applications on multi-core clusters addresses three critical problems that have not been thoroughly investigated in previous studies.

1. In the current trend of computer architecture development, multi-core architecture has become the dominant computing platform. Most HPC machines are clusters of many symmetric multiprocessing (SMP) nodes (multi-core PCs or multi-core servers) connected through high-speed interconnection network. The memory in each node is shared by its cores but is inaccessible by other nodes. Recent studies have shown that a hybrid parallel programming method that applies both shared and distributed memory programming models for this architecture deliveries better performance than parallel methods using only distributed memory model [23], [64], [74], [75], [76]. For FPM, data manipulation in memory is an important factor of mining performance and an efficient use of shared memory for data communication of the jobs running on cores of

the same node can enhance performance. However, most FPM methods designed for cluster computing environment use "shared nothing" parallel model; communication among parallel processes is, therefore, done by message passing [10], [29], [34], [35], [36], [37], [66], [77], [78], [79], [80], [81] , [82], [83], [84]. As a result, benefits of shared memory available within each node is ignored.

2. Load balancing is highly critical for parallel FPM in cluster computing environment. FPM incurs high message passing communication cost making load balancing a great challenge. Irregular and imbalanced computation loads may result in sharp degradation of the overall performance [66]. As shown in Chapter 3, mining without dynamic job scheduling is at least five time slower than static scheduling on shared memory multi-core systems. An efficient workload balancing solution is critical for FPM scalability on cluster architectures.

3. FPM with multiple mining strategies and dynamic detection of switching among the mining strategies based on dataset characteristics is essential for good performance on various types of databases. We derive our FPM design from FEM and DFEM (Chapter 2) and ShaFEM (Chapter 3), and modify it to map best to cluster (combination of shared and distributed memory) computing model.

4.1.2 Contributions

In this chapter, we present a novel parallel FPM algorithm, SDFEM, to address the above-mentioned issues. SDFEM efficiently adapts to the architecture of multi-core clusters and maximizes utilization of the available computing resources. SDFEM is distinguished from prior work due to the following features: (1) exploits the use of shared memory within a node of the cluster, (2) applies multi-level load balancing and (3) uses a

self-adaptive mining approach based on data characteristics. Highlights of our contributions include:

- SDFEM algorithm, a hybrid parallel method utilizing both shared and distributed memory programming models that performs communication within-node via shared memory and between-node via message passing. Using shared memory inter-process communication cuts down the communication cost which is quite high for message passing communication among parallel processes and reduces load balancing overhead. SDFEM also inherits the mining features of ShaFEM (Chapter 3) for faster FPM performance on both sparse and dense data. (Section 4.4 and 4.6)
- A multi-level load balancing method that uses four different strategies: dynamic job scheduling and work sharing for load balancing among cores within a node, and cyclic job scheduling and work stealing for load balancing among nodes in the cluster. The multi-level load balancing method is designed to match the features of hybrid mining model to minimize the overhead of the load balancer as well as to enhance the scalability of FPM (Section 4.5).
- Implementation of the algorithms using MPI (message passing interface) and OpenMP (shared memory), and performance evaluation using real-world datasets used in Chapters 2 and 3 on clusters consisting of multi-core nodes (Section 4.7).

4.2 Background

In this section, we briefly describe multi-core cluster architectures and related issues for development of the FPM for this type of machine.

4.2.1 Architecture of Multi-core Cluster

Multi-core clusters are dominant in high performance computing architecture. From 2002 to 2009, the percentage of top 500 world's fastest computers [85] identified as cluster grew from 18.6% to 83.4% [23]. At the same time, the number of cores within nodes of clusters has been increasing as well. Large-scale applications use HPC machines to support their computationally intensive data analysis. According to an IDC study in 2013, 67% of the HPC sites perform FPM Big Data analysis in their computing systems [86].

Multi-core clusters consist of multiple compute nodes connected by high-speed network infrastructures such as Gigabit Ethernet/Infiniband. Each node consists of several multi-core processors. Cores within a node access the node's shared memory space which may be configured as UMA or NUMA [64]. Cores of a single CPU often have private and shared caches. The resulting architecture is a hybrid memory hierarchy with distributed and shared memory. Figure 4-1 shows an example architecture of a cluster with dual six-core AMD Opteron processors per node sharing main memory; six cores of a processor share L3 cache, each core has private L1 and L2 caches.

The communication latency among cores on a same socket is smaller than that among cores on different sockets due to longer path to cache. The communication latency among cores of different nodes, however, is longest [45]. Using a shared memory programming model, process communication is done by reading and writing shared memory locations. Using a distributed memory model (between nodes for example) messages must be formed by processes and explicitly sent and received between cores of

different nodes. Message formation and communication through the interconnection network incur overhead similar to I/O operations.

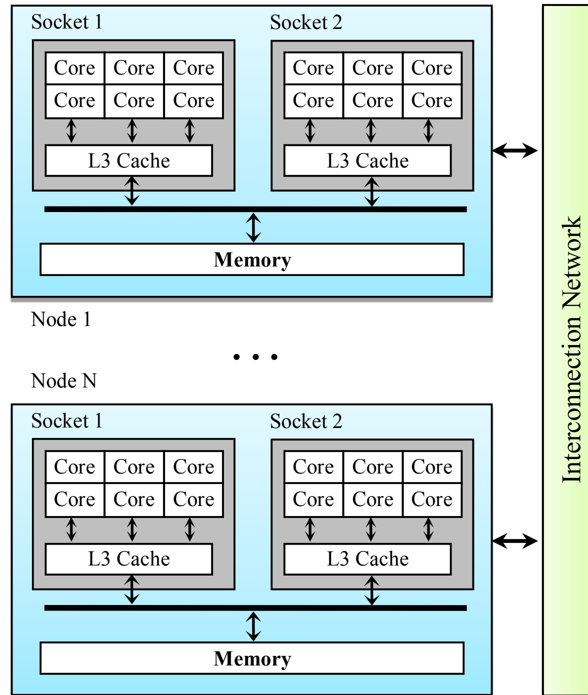
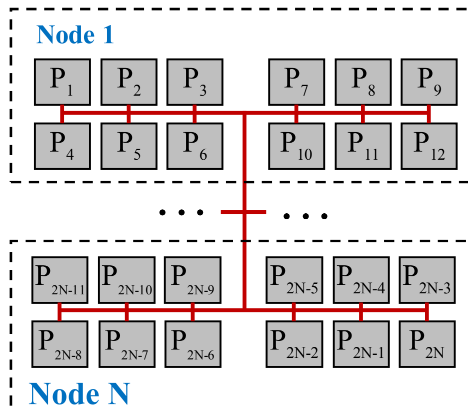


Figure 4-1: Architecture of a cluster with 12-core nodes with dual 6-core sockets

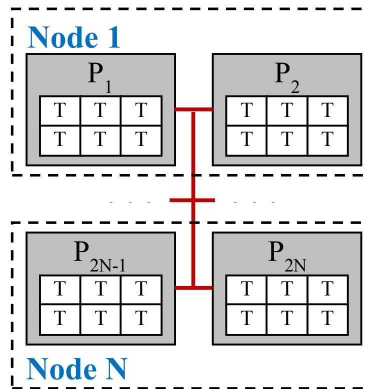
4.2.2 Parallel Programming Models for Multi-core Cluster

Parallel programming models for multi-core clusters require communication and cooperation among processes to be done via interconnection network using messages [22]. MPI (Message Passing Interface) [87] is a message passing parallel programming standard that provides a library interface, including communication routines, to develop parallel programs for distributed memory multiprocessors and assumes all memory is private to parallel processes. Ideally, each process is mapped to run on a processor (one-to-one mapping). Applications developed for clusters commonly use this MPI programming model. Although, MPI programs can run on shared memory multiprocessors they incur message passing overhead.

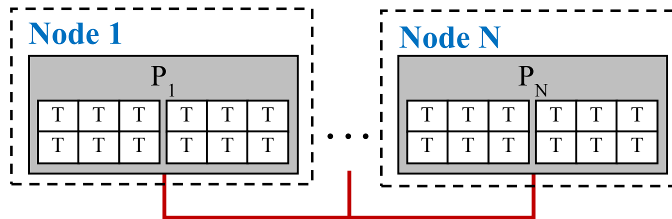
Given the availability of shared memory in each node and increasing number of cores per node, a hybrid model with MPI and OpenMP [65] will provide the means to design applications that can take advantage of features of a cluster architecture [74], [75], [76], [64], [23]. Figure 4-2a illustrates a typical mapping of parallel processes of an MPI program to the cores in a multi-core cluster with 12-core nodes.



(a) MPI (1 process (P) per core)



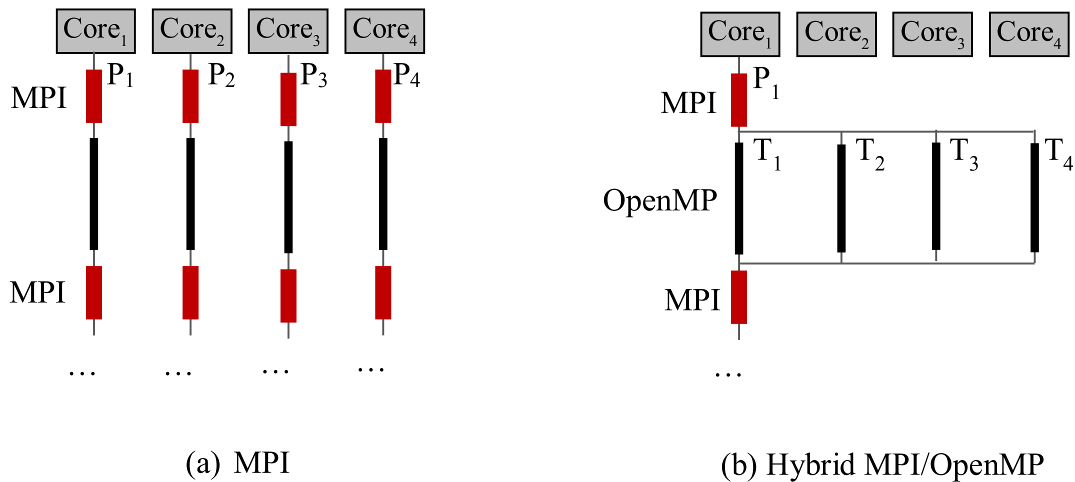
(b) Hybrid MPI+OpenMP
(6 threads (T) per process, 2 processes (P) per node)



(c) Hybrid MPI+OpenMP

Figure 4-2: Mapping of processes and threads to a multi-core cluster

A hybrid MPI–OpenMP program combines both distributed and shared memory programming paradigms and uses MPI for nodes and OpenMP (Chapter 3) for cores of the same node [64]. In this context, we define a process context to be created by MPI which must then fork threads that can be assigned to executed on a core. Execution of hybrid MPI-OpenMP programs starts with invoking a group of MPI processes, each of which forks a number of threads to run in parallel and join upon completion of computation (Figure 4-3b).



(a) MPI (b) Hybrid MPI/OpenMP

Figure 4-3: Execution model of MPI program vs. hybrid MPI/OpenMP

A hybrid program can be mapped onto hardware architecture with different configurations for best performance. For example for the machine with N 12-core nodes of Figure 4-1, one may create a total of N MPI processes where each process can create 12 threads to assigned to the 12 cores of the node (Figure 4-2c). Alternatively, we can create $2N$ processes, 6 threads each, and map 2 processes onto a single node (Figure 4-2b). The best process/thread to core mapping configuration will be the one minimizing communication overhead and taking best advantage of fast shared memory accesses.

For data-intensive applications, especially FPM, this model of programming can provide significant performance gains because it reduces the need for transferring large amounts of data among the processes. The load balancing cost is therefore significantly decreased.

4.3 Related Literature Review

A number of parallel methods have been developed for distributed memory systems [10], [29], [34], [35], [36], [37], [66], [67], [77], [78], [79], [80], [81] , [82], [83], [84]. They have been shown to be efficient in specific contexts. The existing methods do not take advantage of the shared memory within a node. Most methods succeed in reducing data communication and increasing data independence among parallel processes but suffer from load imbalance since their load balancing strategy is heavily based on data partitioning. As a result, they may not scale well in cases like mining with small *minsup*.

Iko et al. [67] proposed a parallel shared nothing FPM method based on FP-growth for PC clusters. It partitions data equally among nodes to construct local FP-trees and deploys a model similar to MapReduce to map conditional pattern bases from send process to receive process. The method utilizes a characteristic called "path depth" to determine the size limit of conditional pattern bases to balance the workload among processes. Work balance is maintained by random selection of receive process for work distribution. This approach is efficient when a good selection of path depth is made. The load balancing strategy with random selection of process is similar to one of load balance strategies in our proposed method. Yu et al. presented a parallel FPM solution for a homogeneous PC cluster [66] based on FP-growth [82]. It uses sampling technique for

load balancing. However, they used synthetic data for experiments and reported poor performance. The method proposed by Tanbeer et el. [37] also based on FP-growth required a single I/O scan only using PP-tree (Parallel Pattern Tree). This study did not described a load balancing strategy and its scalability was only demonstrated for up to 6 processors.

Sucahyo et el. [88] proposed a parallel method based on Eclat for mining dense databases. Database is partitioned into projections, one for each item, and each projection, whose size depends on data characteristics, is stored on the local disk of a node in the cluster. Load balancing is done by distributing the projections to nodes in a round-robin fashion and all nodes follow the same order to decide the destination node to send the conditional pattern bases, which might potentially cause blocking [67]. Similar to the method by Iko [67], O¨zkural et al. presented a parallel solution using vertical data layout [84] and applied a top-down data partitioning scheme in such a way that entire database could be divided into parts with some replications so that they could be mined independently. The data were partitioned to minimize replications and maintain storage balance and computational load. Similar to the FP-growth based methods [37], [82], [67], the benefit of shared memory in multi-core clusters was ignored and good performance was obtained for dense data.

PPS (partial-support-tree) by Souliou et al. [89] was based on a sequential method named partial purport (PS) [90] to compute frequent pattern support. Each process handled a part of database, constructed a local tree and mined frequent patterns from this data structure. Data were distributed to parallel processes using round robin scheme [79]. This method was shown to scale well for dense data. However, when it

mined the test databases with very low *minsup*, its scalability reduced sharply (e.g. its speedup is 1 for Mushroom database with *minsup*=20%).

DPA (Distributed Parallel Apriori) proposed by Yu et al. [66] is one of few FPM methods that parallelized Apriori. Because DPA use a breadth first mining, it is easier to maintain load balancing than in methods using depth first mining strategy. However, this approach requires multiple database scans and suffers from large synchronization overhead because of multiple iterations of the mining loop. Furthermore, Apriori usually has lower performance than most other mining methods; its parallel version (DPA) exhibits a similar poor performance level.

In summary, most existing methods supply their own strategies for data partitioning and job scheduling to balance workload and minimize communication among parallel processes. However, none of them considers the situation where load balance cannot be obtained via work partitioning, particularly when mining with very small *minsup*. This together with the three issues described in Section 4.1.1 motivate the development of a new parallel FPM method.

4.4 SDFEM Algorithm

4.4.1 Overview

SDFEM performs FPM by deploying a group of parallel processes and mapping each process to a multi-core node. Each process creates a group of shared memory threads, mapping each thread to a core. Threads of the same process collaborate to construct the process's local projected XFP-tree which is built from transactions projected to a group of items assigned to its process. Each thread uses this data structure to generate its own frequent patterns. SDFEM finally aggregates frequent patterns

generated by all threads for the final results. SDFEM combines features of both distributed memory and shared memory programming models where between-node communication is done using message passing and within-node communication is done using shared memory. Figure 4-4 illustrates the execution model of SDFEM.

SDFEM model can significantly reduce the overhead of data communication and allow more efficient load balancing. We develop a multi-level load balancing method for SDFEM using four different techniques to increase the CPU utilization and enhance performance. SDFEM performs FPM in two stages

4.4.2 Parallel Projected XFP-tree Construction Stage

Each process constructs a local projected XFP-tree from its data partition. In the first database scan, data is partitioned equally into M*N parts and each part is assigned to a thread where N is the number of processes and M is the number of threads per process. The process and its threads collaborate to compute the global count of all items, identify the frequent items and sort them in frequency descending order. SDFEM distributes the frequent items to processes in cyclic fashion [91]. In this second database scan, each process reponse for entire database; each thread reads a 1/M of database and filters transactions containning the assigned frequent items to construct local projected XFP-tree. For example, if we have two processes, P_1 and P_2, and a list of frequent items a, b, c, d, e then P_1 will mine all frequent patterns ending with item a, c, e and P_2 will mine all frequent patterns ending with b, d. Because of this scheduling method, the projected XFP-tree constructed by each process is much smaller than the complete XFP-tree (Chapter 3). This tree also ensures that each process can work independently. The cyclic scheduling balances the data size of each tree. Figure 4-5, Figure 4-6, Figure 4-7

respectively illustrate the three main steps of constructing projected XFP-trees in FPM for the sample dataset in Table 2-2 and the execution model of 2 processes and 2 threads per process. In Figure 4-7, header table shows items assigned to P_2.

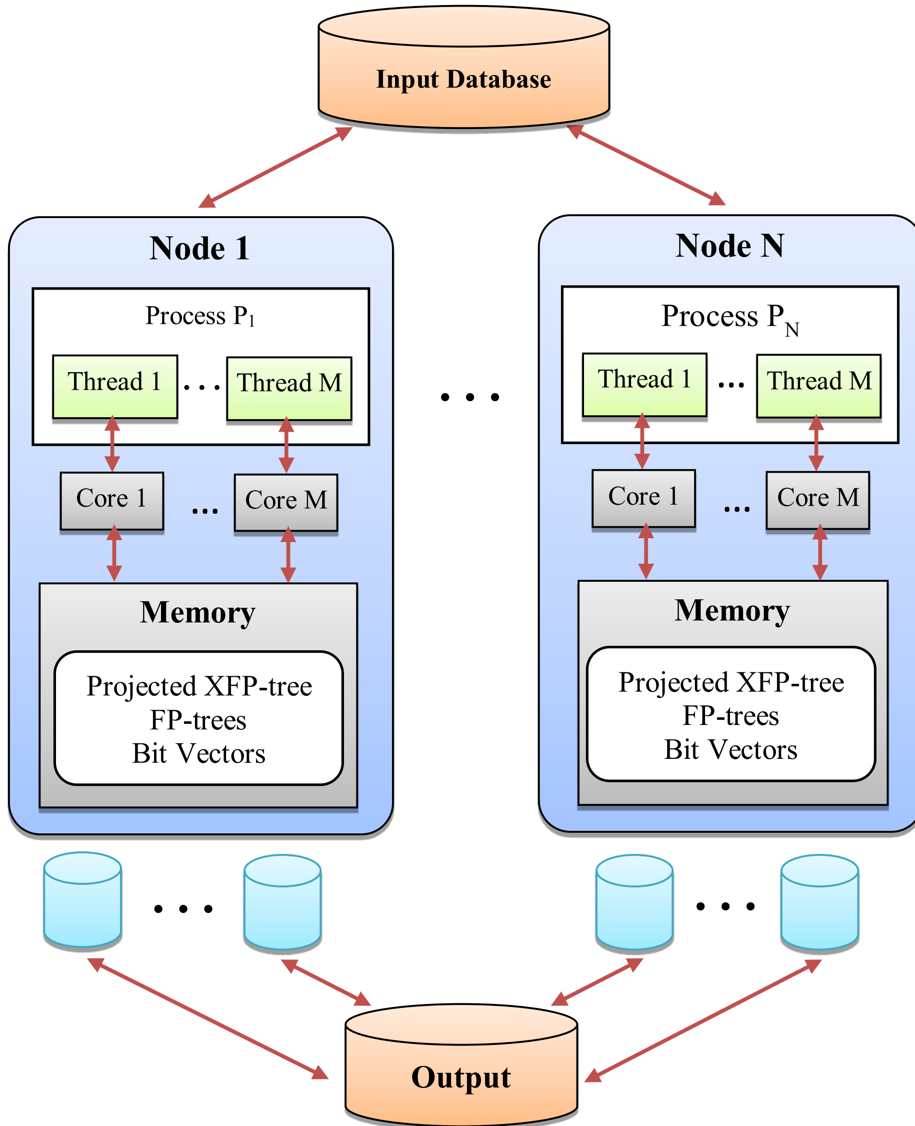


Figure 4-4: Overview execution model of SDFEM

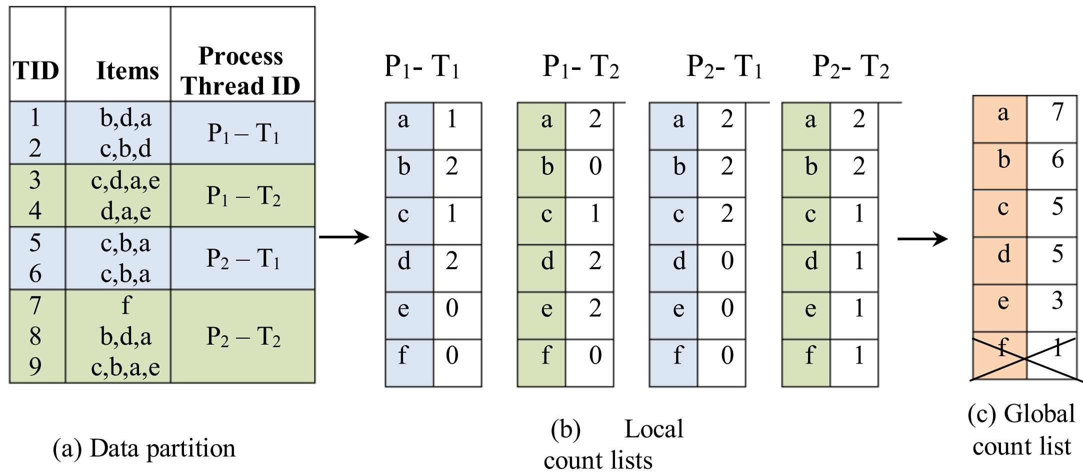


Figure 4-5: Computation of the global count by all processes (P = process; T= thread)

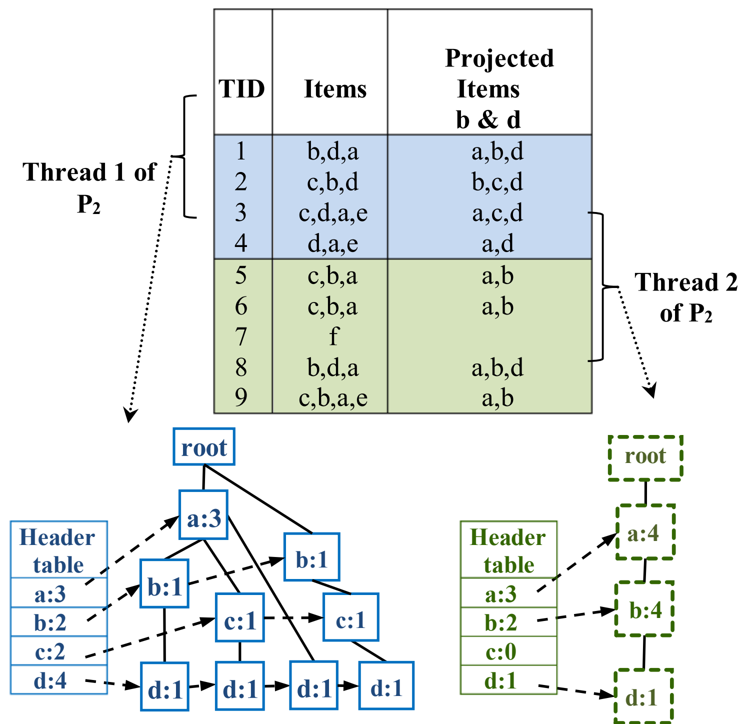


Figure 4-6: Construction of local FP-trees by each thread (T) of Process 2 (P)

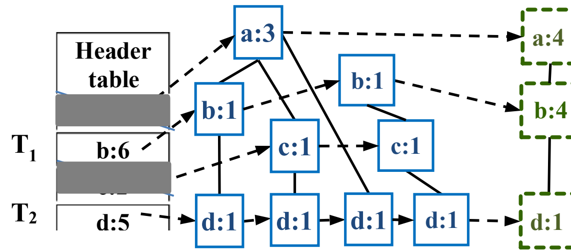


Figure 4-7: The project XFP-tree constructed by Process 2

4.4.3 Parallel Frequent Pattern Generation Stage

After the first stage, each process has a projected XFP-tree in its memory and starts to independently generate frequent patterns using a multi-strategy mining approach similar to the second stage of ShaFEM (Section 3.5, Chapter 3). The mining process includes four tasks *ParallelMinePattern*, *MineFPTree*, *MineBitVector* and *LoadBalancing* (Figure 4-8). Load balancing among processes is necessary because the workload associated with each item or itemset vary depending on the *minsup* input value even with best initial work distribution scheme. Hence, in each SDFEM process, *LoadBalancing* interacts with all threads during the mining process to maintain workload balance.

Similar to ShaFEM, SDFEM inherits the mining features of our sequential method, DFEM, and applies a parallel mining model using two mining strategies *MineFPTree* and *MineBitVector* as described in Chapters 2 and 3. SDFEM conducts the mining model of ShaFEM within each process with several enhancements in data distribution, work partition and load balancing, and scales well on multi-core clusters.

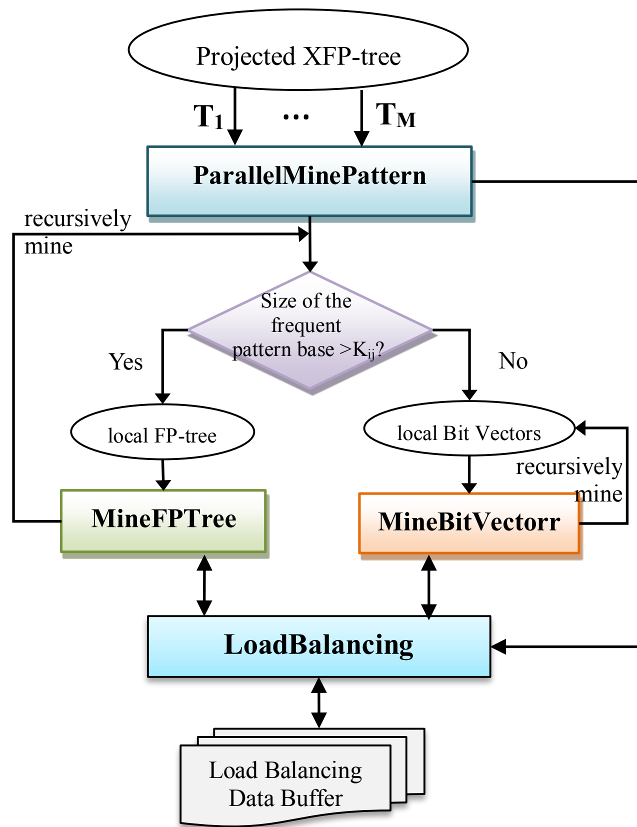


Figure 4-8: The mining model of SDFEM within a process (T=thread)

4.5 Multi-level Load Balancing of SDFEM

SDFEM is designed with two levels of parallelism: thread parallelism on shared memory multicores, and process parallelism on cluster nodes, which require minimizing communication for scalability. Therefore, load balancing in SDFEM also includes two levels: within-node load balancing for threads and between-node load balancing for processes. Balancing workload in a parallel task that can be done implicitly with job scheduling and explicitly with load balancing techniques such as work sharing or work stealing. To maximize workload balance, we apply four load balancing techniques in SDFEM (Table 4-4). For simplicity, we implement a single data structure called load

balancing data buffer (LBDB) used by *LoadBalancing* for both within-node and between-node load balancing purposes.

Table 4-1: Load balancing techniques applied in SDFEM

	Within-node load balancing	Between-node load balancing
Implicit techniques	Dynamic Job Scheduling	Static Cyclic Scheduling
Explicit techniques	Work Sharing	Work Stealing

4.5.1 Within-node Load Balancing

SDFEM applies dynamic job scheduling and work sharing to maintain the workload balance and optimal CPU utilization among the threads of the same process.

4.5.1.1 Dynamic job scheduling

Threads of the same process dynamically obtain the next available item from the header table of the projected XFP-tree and complete mining all frequent patterns ending with this item. This dynamic scheduling scheme, ParallelMinePattern, is implemented the same way as in ShaFEM (Chapter 3). In OpenMP, this is implemented by simply defining dynamic directive for the parallel loop.

4.5.1.2 Work sharing

Dynamic job scheduling is efficient. It however does not ensure load balance for cases where number of frequent items is considerably small, dense databases for example. Hence, a load balancer is added to each process; it maintains a load balancing data buffer holding shared data subsets. Busy threads within a process share their

workload. Available threads seek additional work from this buffer. Because this data structure is shared among threads of the same process, it can easily be updated by all threads within a process. Figure 4-9 illustrates the load balancing using work sharing.

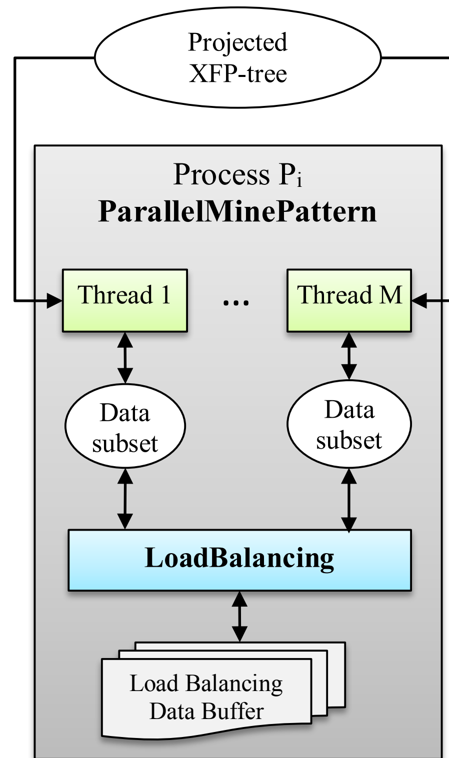


Figure 4-9: Within-node load balancing with work sharing

During the frequent pattern generation stage, threads periodically check this data buffer via *LoadBalancing*. If it is empty or if the number of data subsets is smaller than a certain limit Max_{LBDB}, threads will add their newly generated data subsets to the buffer, which are either FP-tree or Bit Vectors depending on the mining strategy being used (i.e. *MineFPTree* or *BitVector*). When a thread completes mining its data, it increments its counter, *TC*, by one. It then checks the data buffer to take a data subset and recursively mines this data subset. If the buffer is empty, the thread will wait until new data subsets are added, or until process status changes to *terminating*. This load balancing method ensures that threads of the same process remain busy until they are all done with

generating frequent patterns for the process's data partition. To minimize memory usage and the overhead of maintaining data buffer, we keep the number of buffer entries as small as possible. For example, in our experiments, the maximum number of buffer entries was set to the total number of threads of all processes.

4.5.2 Between-node Load Balancing

SDFEM applies static cyclic job scheduling and work stealing techniques to balance workload among the processes.

4.5.2.1 Cyclic job scheduling

Due to the large amount of processed data, dynamic job scheduling may result in huge communication overhead and bottleneck. We apply static job scheduling to distribute work among processes because it is simple, has practically no overhead and does not require communication and synchronization optimization. In the first stage of projected XFP-tree construction after the frequent items are found and sorted in frequency descending order, they are distributed to processes in a cyclic fashion as illustrated in Stage 1, Section 4.4. Process filters database using the assigned items to construct XFP-tree.

4.5.2.2 Work stealing

In most cases when the number of frequent items are large, cyclic job scheduling is good for initial work partition among parallel processes. We apply work stealing to maintain better load balance, especially for ~~the~~ cases where mining workload is associated with significantly varied frequent items or the number of frequent patterns is small. Based on work stealing techniques, idle processes actively look for busy processes

to request for more work. Since only idle processes attempt to communicate, the amount of communication is reduced and the overhead is well tolerated compared to idle time of processes without work [78]. Both work sharing and work stealing use the same data buffer.

On starting frequent pattern generation, each process keeps a process status list whose elements indicate the status of all processes (i.e. *working*: a process is still generating frequent patterns from its pre-schedule data, *balancing*: a process completes its work portion and is requesting for job from a remote process, *terminating*: a process completes its work and there is no *working* process to request for job) and are initialized with *working* status. If a process P_k completes generating all frequent patterns from its projected XFP-tree (i.e. a *balancing* process) it will pick up a victim process P_h among the *working* processes and sends a job request to P_h. Figure 4-10 depicts the load balancing model with work stealing between P_k and P_h.

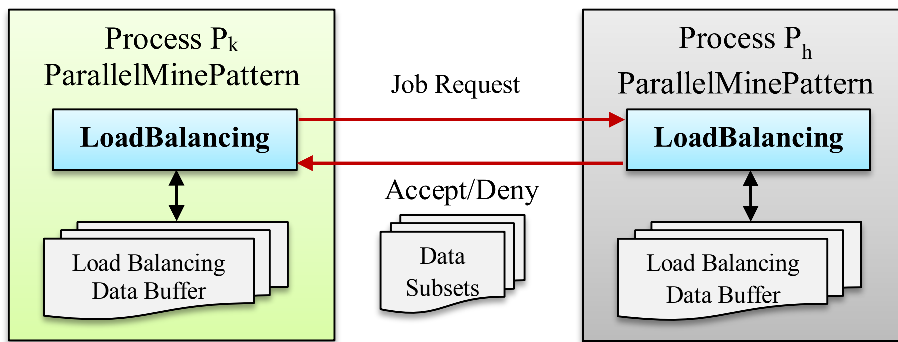


Figure 4-10: Between-node load balancing with work stealing

Upon receiving the request, P_h can either accept or deny in following scenarios:

- If P_h is *working* and its load balancing data buffer (LBDB) has more than M subsets where M is the number of parallel threads of P_h, it will reserve M data

subsets, pack the remaining ones and send them to P_k together with an acceptance notice. P_k receives the data sent by P_h, puts them into its LBDB and continues its mining process on the received data.

- If P_h is *working* and its LBDB has less or equal than M subsets, P_h sends to P_k a denial notice. P_h always keeps at least M data subsets in its LBDB to ensure that the between-node load balancing does not interrupt the within-node load balancing process. When P_k receives the denial response, it will select another *working* process in its list to ask for job.
- If P_h is *balancing*, it will send a denial response together with its status to P_k. When P_k receives the response, it will update the status of P_h in its process status list to avoid requesting P_h for job in future and look for another randomly selected *working* process to request for job.
- If a *balancing* process P_h cannot find a *working* process in its list because all processes are marked as either *balancing* or *terminating*, its status will switch to *terminating* by sending a broadcast to all other processes to inform them of its status and exits the frequent pattern generation stage. The other processes update status of P_h in their list.

Although work stealing technique requires communication among processes, its overhead is always small because of the following reasons. First, SDFEM employs the hybrid programming model that create fewer parallel processes (i.e. usually equal to the number of nodes in the cluster). Because the number of processes involved in load balancing is small and data communication is decentralized due to work stealing, the overhead of load balancing is much smaller than that of parallel programming with pure

message passing. Second, in SDFEM, only one thread, the master thread, of each process participates in work stealing while the other threads continue its mining work.

4.6 Algorithmic Description

SDFEM algorithm (Figure 4-11) consists of five sub algorithms ParallelMinePattern2 (Figure 4-12), MineFPTree4 (Figure 4-13), MineBitVector2 (Figure 4-14) and LoadBalancing (Figure 4-15), UpdateK3 (Figure 4-16). SDFEM starts by creating a group of processes; each process then creates a group of threads and executes the SDFEM algorithm (Figure 4-11).

SDFEM algorithm
Input: Transactional database *D* and *minsup*
Output: Complete set of frequent patterns
1: Threads scan their partitions of *D* and cooperate to find all frequent items
2: Threads re-scan *D* to construct projected XFP-tree X*T* (1 tree each process)
3: $N = 9$, $Step = 32$, *items* = the number of frequent items in *D*, $TC_i = 0$
4: $K_{ij} = 0$ where K_{ij} is private to each thread, *i* is process id and *j* is thread id
5: Each thread calls **UpdateK3**(*items*, $N*Step$)
6: Each thread calls **ParallelMinePattern2**($XT, \varnothing, minsup, TC_i$)

Figure 4-11: SDFEM algorithm

ParallelMinePattern2 algorithm (Figure 4-12) is invoked after each process completes constructing its projected XFP-tree, which is private to the process and shared among threads.

```
ParallelMinePattern2 algorithm
---------------------------------------------------------------
Input: shared projected XFP-tree XT_i , minsup , thread counter TC_i,
Output: frequent items
1:  K_ij = 0 where i is process id and j is thread id
2:  Flag = true
3:  Parallel Self-Scheduled For each item α assigned to Process i
4:  {   Output α
5:      Size_C = the size of α's conditional pattern base C
6:      Compute and update threshold K_ij
7:      If  Size_C >K_ij  Then
8:          Construct α's local conditional FP-tree T
9:          Call MineFPTree4(T,α,minsup)
10:     Else
11:         Construct α's  local bit vectors V and weight vector w
12:         Call MineBitVector2(V,w,α,minsup)
13:     End if
14: }
15: TC_i = TC_i + 1 (*using atomic operation*)
16: While Flag = true
17:     Flag = LoadBalancing (∅, TC_i )
18: End while
```

Figure 4-12: ParallelMinePattern2 algorithm

MineFPTree4 and MineBitVector2 algorithms (Figure 4-13 and Figure 4-14) mine frequent patterns using the similar approach presented in Chapter 3. However, these

algorithms perform additional load balancing task using *LoadBalancing* algorithm (Figure 4-15).

```
MineFPTree4 Algorithm

Input: FP-tree T, suffix, minsup, thread counter TC_i
Output: set of frequent patterns
1:  If T contains a single path P then
2:  For each combination x of the items in P
3:          Output β = x ∪ suffix
4:          Compute and update threshold K_ij
5:  Else Foreach item α in the header table of FP-tree T
6:          Output β = α ∪ suffix
7:          Size = the size of α's conditional pattern base C
8:          DS = {C, β}
9:          Flag = LoadBalancing(DS, TC_i)
10:           If Flag=false then
11:               Compute and update threshold K_ij
12:               If Size > K_i Then
13:                   Construct α's local conditional FP-tree T'
14:                   Call MineFPTree4(T',β,minsup,TC_i)
15:               Else
16:                   Construct α's local bit vectors V and w
17:                   Call MineBitVector2(V,w,β,minsup,TC_i)
18:               End if
19:           End if
20:  End if
```

Figure 4-13: MineFPTree4 algorithm

MineBitVector2 algorithm
Input: Bit vectors V, weight vector w, $suffix$, $minsup$, thread counter TC_i
Output: Set of frequent patterns
1: Sort V in support-descending order of their items
2: **For** each vector v_i in V
3: { Output β = item of $v_i \cup suffix$
4: **For** each vector v_j in V $with\ j < i$
5: { $u_j = v_i$ AND v_j
6: sup_j = support of u_j computed using w
7: **If** $sup_j \geq minsup$ **Then** add u_j into U
8: }
9: **If** all u_j in U are identical to v_i
10: **Then For each** combination x of the items in U
11: Output $\beta' = x \cup \beta$
12: **Else If** U is not empty **Then**
13: $DS = \{U, \beta\}$
14: $Flag$ = **LoadBalancing**(DS, TC_i)
15: **If** $Flag$=$false$ **then**
16: Call **MineBitVector2**($U,w,\beta,minsup$)
17: **End if**
18: **End if**
19: }

Figure 4-14: MineBitVector2 algorithm

LoadBalancing algorithm (Figure 4-15) performs work sharing and work stealing as described in Section 4.5. All threads involve in work sharing (Line 2 – 16) while only one thread of a process perform work stealing (Line 17 – 29). Line 4, 8, 24 are performed using a synchronization method such lock or critical section as many threads can update $LBDB_i$ at the same time.

```
LoadBalancing algorithm
Input: data subset DS, thread counter TC_i,
Output: Flag
(* Create shared load balancing data buffer LBDB_i & process status list PSL_i *)
1:      Flag = false
2:      Size_LBDBi = the number of conditional pattern in C
3:      If C ≠ ∅ and size < Max_LBDBi then
4:          Add C and its suffix into LBDB_i critical section/lock
5:          Flag = true
6:      End if C = ∅ then
7:      If Size_LBDBi > 0 then
8:              Extract DS = {C, suffix} from LBDB_i critical section/lock
10:             If DS was created by MineFPTree4 previously Then
11:             {    Construct conditional FP-tree T from C
12:                  Call MineFPTree4(T, suffix, minsup, TC_i)  }
13:             Else
14:                  Call MineBitVector2(C,w, suffix,minsup, TC_i)
15:             End if
16:     End if
17:     If Thread id = 0 then
18:           Check and serve a work stealing request from a remote process
19:           Check for status message from a remote process and update PSL_i
20:           Update process status based on TC_i , Size_LBDBi, PSL_i
21:           If status = loadbalancing then
22:           {     Send a work stealing request to a remote working process
23:                 Wait and receive response from the remote process
24:                 If request is accepted then
25:                      Add received data into LBDB_i using critical section/lock }
26:           Else if status = terminating then
27:           {     Send status message to other processes inform its status
28:                 Flag = true   }
29:     End if
```

Figure 4-15: LoadBalancing algorithm

Similar to DFEM and ShaFEM, SDFEM use switches between the two mining strategies based on a heuristic that uses a threshold K. Each parallel thread maintains its own K_{ij} and measures K_{ij} based on the locally processed data where i is process id and j is thread id. UpdateK3 algorithm (Figure 4-16) is used to update K_{ij} as mining proceeds.

UpdateK3 Algorithm
Input: NewPatterns and Size *Output*: Updated values of K_i (*Initialization for the first call to UpdateK3: Create a private array X with N elements, Set all $X[h]$ to zero *) 1: **For** $h = 0$ to $N - 1$ 2: **If** $Size > h*Step$ **then** $X[h] = X[h] + NewPatterns$ 3: **Else** Exit Loop 4: $K_{ij} = 0$ 5: **For** $j = N\text{-}1$ to 1 6: **If** $X[h\text{-}1] \geq 2*X[j]$ **then** $K_{ij} = (h+1)*Step$ and Exit Loop

Figure 4-16: UpdateK3 algorithm

4.7 Performance Evaluation

4.7.1 Experimental Setup

Datasets: The five real datasets from Chapter 2 and 3 are used for our experiments: two sparse, one moderate and two dense databases obtained from the FIMI Repository [59], a well-known repository for FPM. The database features are reported in Table 4-1.

Software: SDFEM can be implemented using different programming platforms like OpenMPI [92]. MPICH [93], LAM/MPI [94], which are different implementations of MPI, as well as OpenMP, Pthread or other multi-threading platforms. In our experiments, we choose OpenMPI and OpenMP to implement SDFEM.

Table 4-2: Experimental datasets of SDFEM

Dataset	Type	# of Items	Average Length	# of Trans.
Chess	Dense	76	37	3196
Pumsb	Dense	2113	74	49046
Accidents	Moderate	468	33.8	340183
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Hardware: We use a cluster of many Altus 1702 machines where each node is equipped with dual AMD Opteron 2427 processor, 2.2GHz, 24GB memory and 160 GB hard drive. In our experiments, the benchmarks of SDFEM with up to 120 cores (i.e. 10 nodes of our cluster) were conducted. The operating system is CentOS 5.3, a Linux-based distribution.

4.7.2 Execution Time

We demonstrate the performance of SDFEM by measuring its execution time for various number of cores of the test cluster on six datasets. The sequential test mode was done on 1 core. The parallel one was done by running the program on varying number of nodes from 1 to 10, each node runs up to 12 cores, providing a range of 12 to 120 threads or cores (Figure 4-2c). The test results show that the sequential execution times of SDFEM are close to DFEM. Hence, the overhead of parallel implementation of SDFEM is small. Experimental results of SDFEM with various number of cores are shown in Figure 4-17.

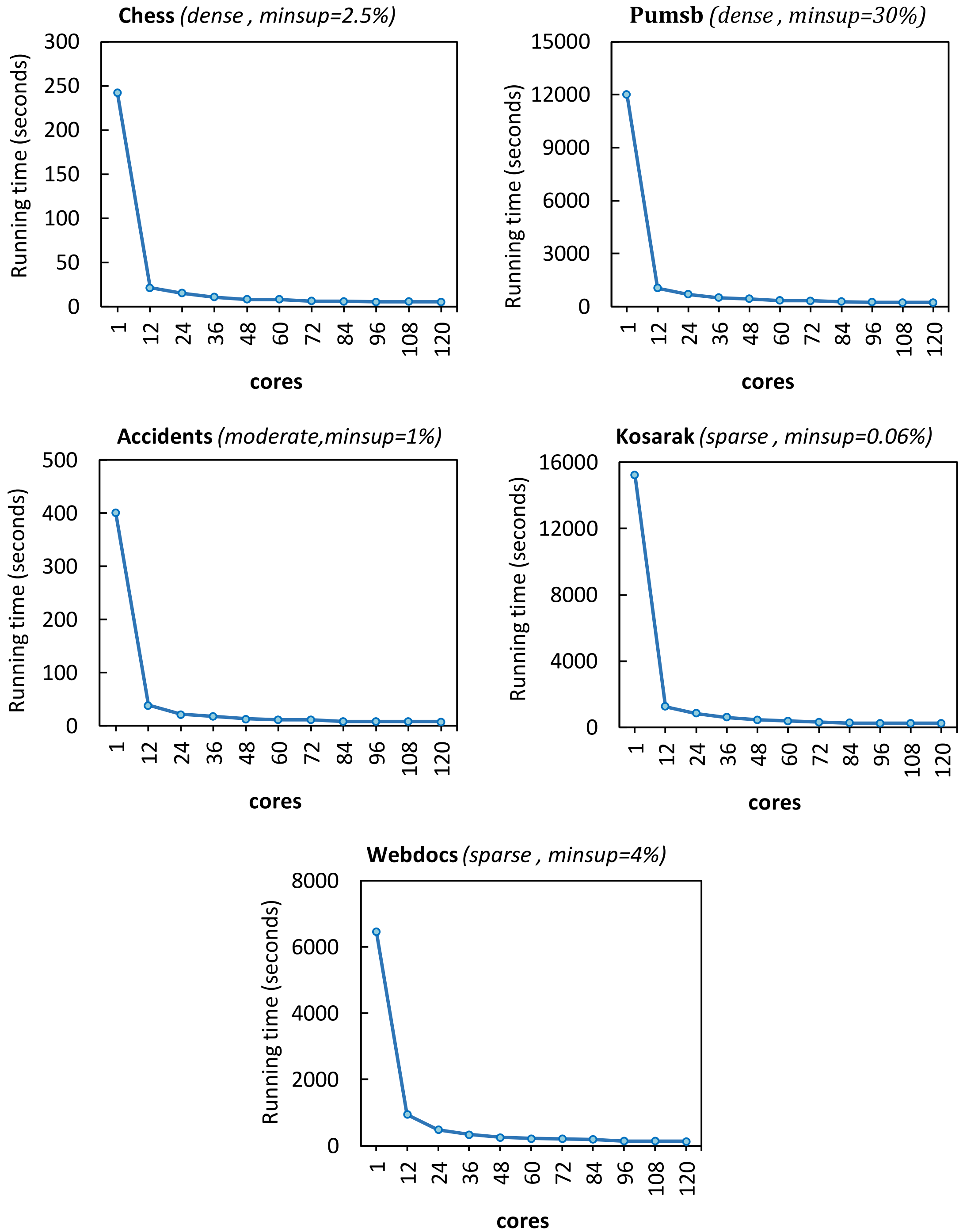


Figure 4-17: Running time of SDFEM (from 1 to 120 cores)

The results show significant reduction in execution time is obtained for all test cases. In the experiments, SDFEM reduces the mining time on Webdocs databases from

6460 seconds on 1 core to 130 seconds on 120 cores which saves 97.98% of time that sequential execution requires (i.e. 97.98% = (6460-130)/6460*100%). The time of SDFEM on Kosarak database is cut down 98.4% (from 15234 seconds on 1 core to 238 seconds on 120 cores). Similarly, the percentage of execution time savings for Chess, Pumsb and Accidents are 97.79%, 98.08%, 97.95% respectively. This performance improvement has come from sharing mining workload for large number of cores and reducing the amount of data that each parallel process/thread has to handle. Because SDFEM inherits the mining features of DFEM, its performance gain is consistent for both dense and sparse databases.

4.7.3 Speedup

To evaluate scalability of SDFEM to the size of cluster, we compute its speed up by dividing the sequential time of SDFEM by the parallel execution time (i.e. for 12, 24, 36,…120 cores) and present the results in Figure 4-18. We can see that for most cases, speed up increases when the number of cores is increased. Speed up values of six datasets on 120 cores are 45.4 (Chess), 52.1 (Pumsb), 64.8 (Kosarak), 48.7 (Accidents), 49.6 (Webdocs). Many factors limit scalability of most parallel FPM methods like synchronization, load balancing and data communication overheads or limitations of test hardware like serial I/O. It is important to note that, as the number of nodes (multiples of 12 in plots of Figure 4-18) is increased, higher speedups are obtained which shows SDFEM scales for lager machines.

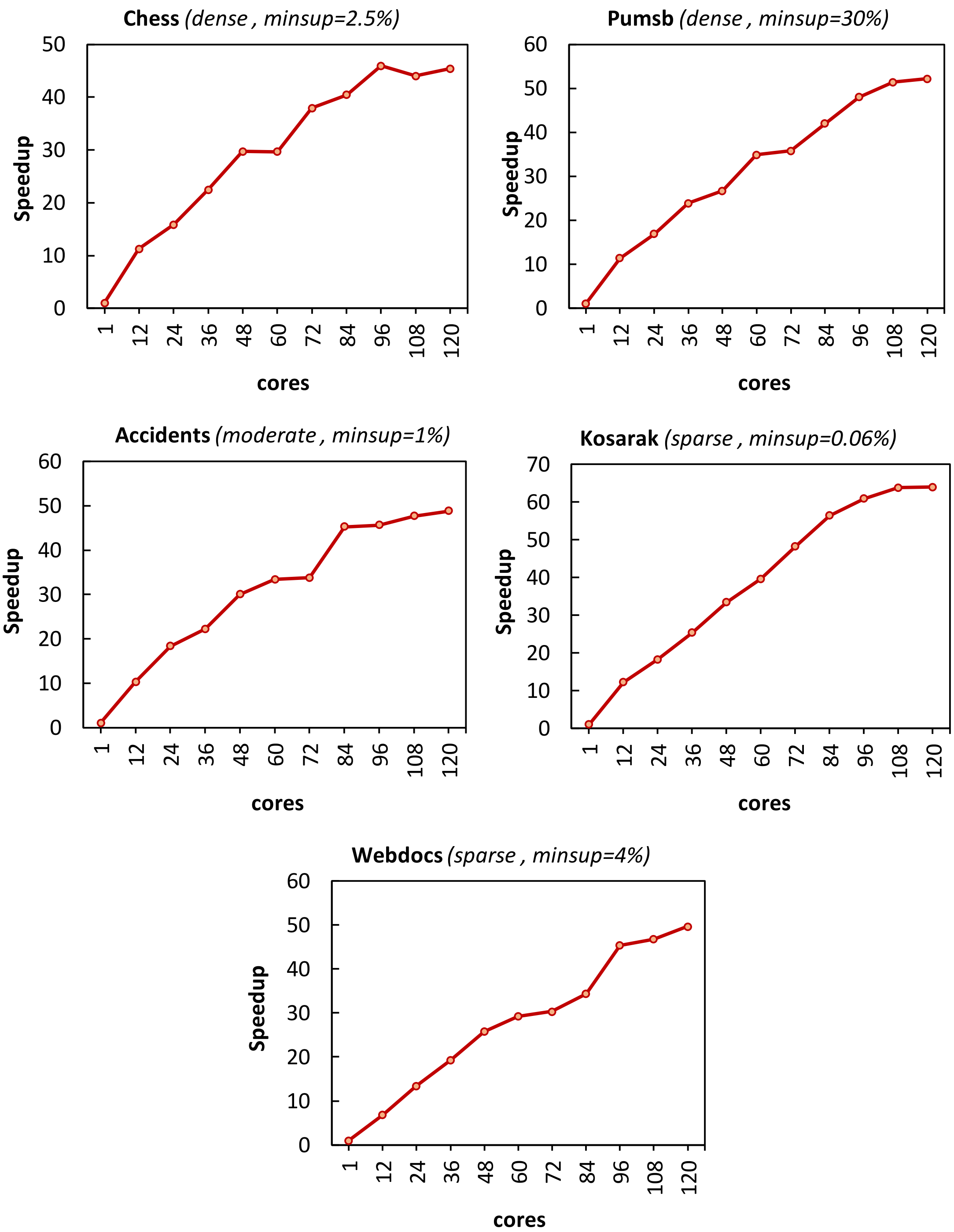


Figure 4-18: Speedup of SDFEM (from 1 to 120 cores)

As mentioned in the previous chapters, performance of FPM is significantly impacted by the *minsup* value; the smaller *minsup*, the larger execution time. That is because more transactions are load into memory in the second database scan and more candidates statisfy the condition to be frequent pattern. Therefore, we also examined scalability of SDFEM when the *minsup* input value varies. Figure 4-19 presents running time (for 1 core and 120 cores) and speedup ($Time_{1\ core}$ / $Time_{120\ cores}$) of SDFEM on Webdocs database (our experimental results on the other databases show similar trend). According to these results, as *minsup* values decrease, execution time of SDFEM also increases for both test cases with 1 core and 120 cores. Furthermore, speedup increases when data are mined with smaller *misup*. This feature is important as it shows mining with small minsup scales well on Big Data.

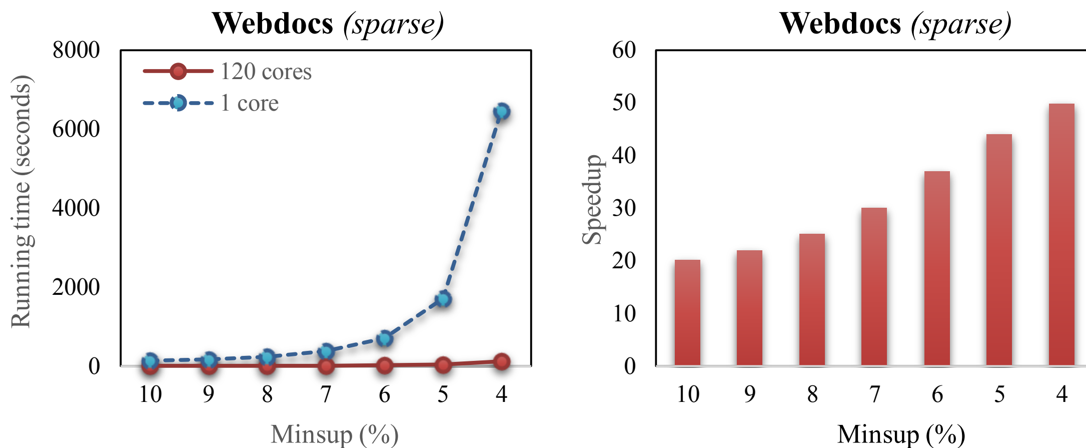


Figure 4-19: Running time and speedup of SDFEM when minsup varies

4.7.4 Impact of Hybrid MPI-OpenMP Programming Model

Application of hybrid MPI-OpenMP programming model is an important feature of SDFEM, making it distinguished from related work. We study the impact of this programming model in comparison with the traditional pure MPI model. For this

purpose, SDFEM has been benchmarked using 5 compute nodes with 60 cores in total in two scenarios (as depicted in Table 4 3):

Scenario 1- 1 process per core and total processes = 60

Scenario 2- 12 threads per process, 1 process per node and total threads = 60

In Scenario 1, SDFEM performs exactly like a MPI program. In Scenario 2, SDFEM applies the hybrid programming model presented in Sections 4.2.2 and 4.4. The comparison results in Table 4-3 shows clear evidence that using hybrid model can improve significantly FPM performance. Compared to pure MPI, the hybrid mode with 12 threads per process enhance 70.0% up to 329.5% of mining performance. For Kosarak, SDFEM with the hybrid model runs much faster than its pure MPI version (474 seconds vs. 2036 seconds).

Table 4-3: Time comparision of pure MPI vs. hybrid MPI-OpenMP (60 cores)

Datasets	**MinSup**	**Scenario 1** Pure MPI (1) (sec.)	**Scenario 2** Hybrid (2) (sec.)	**Time Difference** (3)=(2)-(1)	**Performance Improvement** (4)=(3)*100/(2)
Chess	2.50%	42	11	31	281.8%
Pumsb	30%	128	68	60	88.2%
Accidents	1%	25	14	11	78.6%
Kosarak	0.06%	2036	474	1562	329.5%
Webdocs	4%	436	220	216	98.2%

4.7.5 Impact of Different Load Balancing Techniques

Another important factor impacting FPM performance is load balancing. We evaluate the efficiency of the four load balancing techniques applied in SDFEM (Section 4.5) by implementing 4 different versions of SDFEM where each integrates a combination of different techniques as listed in and benchmarking them using 120 cores (12 threads/cores per process, 1 process per nodes). In Table 4-4, the underlined values

indicate load-balancing techniques in Section 4.5. All load balancing techniques are applied in SDFEM-V4 while fewer are used in the others: SDFEM-V1 (1 technique), SDFEM-V2 (2 techniques) and SDFEM-V3 (3 techniques). The test results presented in Table 4-5 indicate that SDFEM-V4 runs much faster the other three versions, showing clearly the importance of load balancing techniques.

Table 4-4: Four versions of SDFEM with different load balancing techniques

Techniques	**SDFEM-V1**	**SDFEM-V2**	**SDFEM-V3**	**SDFEM-V4**
Within-node job scheduling	Static (cyclic)	Dynamic	Dynamic	Dynamic
Within-node load balancing	N/A	N/A	Work Sharing	Work Sharing
Between-node job scheduling	Static (cyclic)	Static (cyclic)	Static (cyclic)	Static (cyclic)
Between-node load balancing	N/A	N/A	N/A	Work Stealing

Table 4-5: Running time of four versions of SDFEM (120 cores)

Datasets	**MinSup**	**SDFEM-V1** (sec.)	**SDFEM-V2** (sec.)	**SDFEM-V3** (sec.)	**SDFEM-V4** (sec.)
Chess	2.5 %	42.2	42.3	7.5	5.2
Pumsb	30 %	3153	3152	341	252
Accidents	1 %	44	27	9.7	8.1
Kosarak	0.06 %	2153	2020	304	248
Webdocs	4 %	637	316	130	129

We measure speed up of the four versions by dividing the sequential execution time of SDFEM by the parallel time of each version. Figure 4-20 shows that the performance of SDFEM is enhanced for each added technique. Among those, applying of within-node load balancing using work sharing introduces the most significant performance and speed up gain compared to the other techniques on most test databases because we observe the large increase in speed up (SDFEM_V3). This also demonstrates

that use of shared memory not only reduces data communication but also improves workload balance by providing an environment suitable for work sharing technique. The impact of other techniques vary depending on the database. For example, sharp increases in SDFEM on Chess, Pumsb and Kosarak are observed with work stealing (SDFEM_V4).

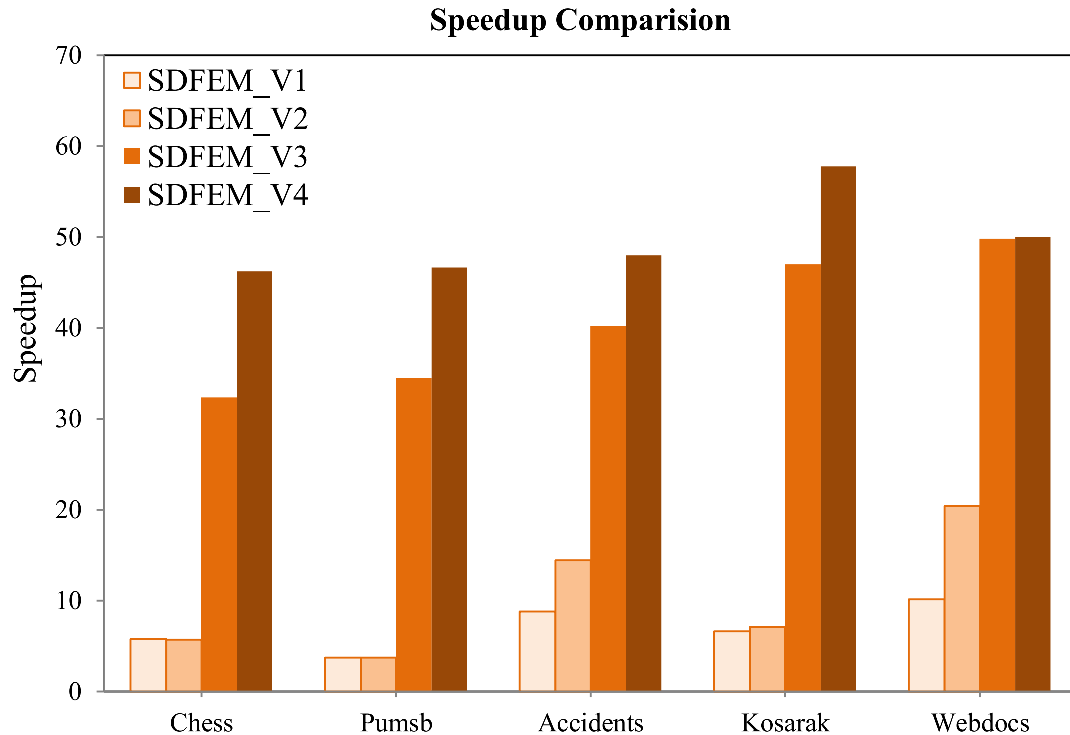


Figure 4-20: Speedup of four versions of SDFEM compared to its sequential time

4.8 Conclusion

We present SDFEM, a novel parallel FPM for multi-core clusters, as a high performance FPM solution for large-scale applications. SDFEM has three main features which have not been investigated by the prior parallel work. They include (1) use of hybrid programming model to leverage the benefits of shared memory and enhance the mining performance; (2) application of multiple load balancing techniques to achieve high performance and scalability; and (3) utilization of data characteristics based mining

approach, that we developed and presented in Chapter 2, to perform efficiently on different types of data. Our performance evaluation has shown that in our test cases, SDFEM can save 97.79% - 98.4 % compared to sequential time. SDFEM on 120 cores of our cluster run 45.4 – 64.8 times faster its sequential time for the test datasets.

5. Parallel Frequent Pattern Mining on GPU

5.1 Introduction

General-Purpose Computing on Graphics Processing Units (GPGPU, referred to as GPU in this Chapter) have emerged as powerful computing resources for general–purpose computing applications. They have been increasingly used as co-processors to provide parallel processing capability that was once dominated by CPUs [95], [96]. GPGPU applications can be developed by using either CUDA [97] or OpenCL [98].As the volume of data generated in most fields is fast growing, applying high performance techniques to enhance the overall performance of FPM has become important.

5.1.1 Motivation and Related Literature

Modern GPUs are designed with up to hundreds of computing units to be able to process a massive number of data elements in parallel at very high speed. GPUs support SIMT, a Single Instruction Multiple Thread model of computation, where groups of concurrent *thread*s of execution created through a *kernel* launched from CPU on GPU will execute their assigned instruction streams in parallel. GPUs work well for applications exhibiting regular patterns. Most efficient FPM algorithms, which are designed and optimized for CPU, use complex data structures and do not lend themselves well to be ported to GPU.

Most existing methods for GPU [99], [100], [101], [102], [103], [104], [105], [106], [107], [108] are based on Apriori because they use breadth-first strategy and candidate generation-and-test approach to create a large amount of workload with data suitable for presentation on GPU and are easily parallelizable. However, for very large

databases, the Apriori-like methods are significantly slower and consume much more memory in comparison to sequential methods like Eclat [31], FP-growth [25], and specially our proposed methods, FEM [39] and DFEM [40]. In many cases, Apriori-like methods are hundred times slower than FP-growth or its variants like the FP-tree traversal time [26], H-mine [27], nonordfp [28], the use of FP-array data structure [29] and FP-growth with database partition projection [30]. This means that for certain databases, even the best GPU Apriori-like algorithm using hundreds of parallel computing units may perform at best as well as some sequential methods running on a single CPU. While Apriori-like methods have been shown to perform efficiently on sparse data they do not perform well on the dense databases [59], [32], [48]. Similarly, these GPU based Apriori methods for FPM exhibit the same poor performance on sparse datasets as well.

5.1.2 Contributions

We propose CGMM (CPU & GPU based Multi-strategy Mining), an extended version of DFEM which utilizes both CPU and GPU for its computation. CGMM works on the machine equipped with GPU and is an alternative method for DFEM when mining with low *minsup* values. The following features of CGMM contribute to its improved performance and distinguish it from the prior work:

1. CGMM consists of two different mining strategies, *CPUBasedMining* and *GPUBasedMining*, specifically designed to exploit the computing power of CPU and GPU. It inherits the mining feature of FEM that uses a heuristic approach to dynamically select a suitable strategy for each data subset of the database based on its density characteristics during the execution.

2. The *CPUBasedMining* strategy uses only CPU to mine the frequent patterns by recursively constructing FP-trees without generating a large number of candidates. It is applied for data portions with sparse characteristics.
3. The *GPUBasedMining* strategy uses GPU as the main computing engine to mine the data portions with dense characteristics using a hybrid solution that consists of a new adaptive breadth-first approach, bit vector data structures, and candidate generation and test approach.

5.2 Background

5.2.1 GPU Architecture

GPU has becomes a popular computing device and is embedded in most computer systems (e.g. mobile devices, PCs, servers, cluster, etc.) [101]. They are broadly known as the special type of processor used to accelerate graphics applications. Figure 5-1 depicts a typical design of computer systems equipped with GPU devices. In this architecture, one or more GPUs (i.e. co-processors referred as *device*) connect with CPUs (main processors referred as *hosted*) via peripheral component interconnect express (PCI-e).

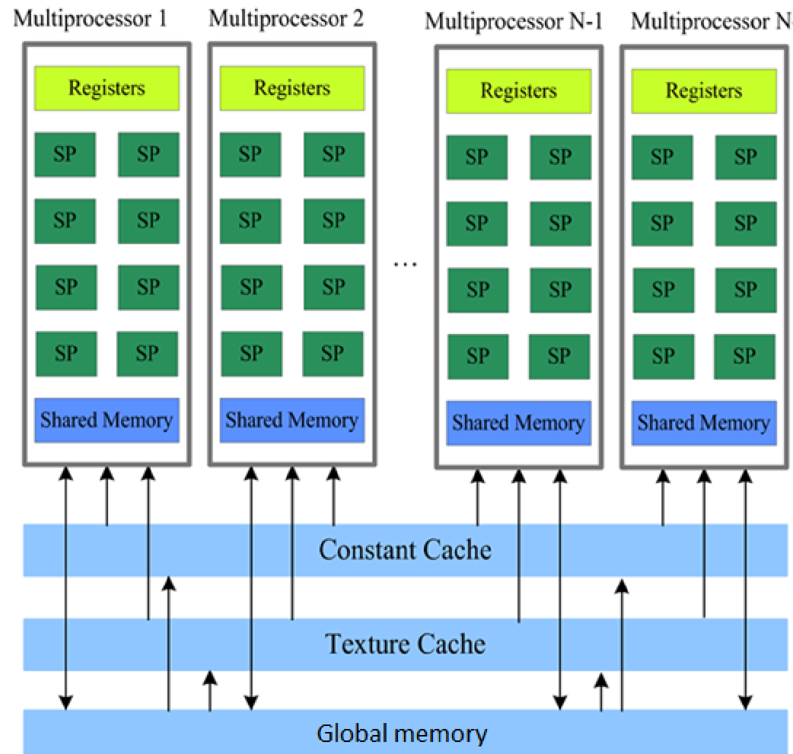


Figure 5-1: The architecture of Nvidia's GPU

Modern GPUs have between dozens to hundreds of computing units/cores and can deliver much larger performance than a CPU for the right type of application. Their architectures typically consist of several streaming multiprocessors (SM) sharing same device memory. Each SM can have eight or more computing units/cores (most often referred to as streaming processor – SP) depending on the device model. Nvidia's GPUs with CUDA architecture are dominantly used at the time of this writing. Development of none-graphics applications for GPUs have become easier by programming tools like CUDA [97], OpenCL [98], Access 3. The architecture of a CUDA-enabled GPU consist of one or many stream multiprocessors (SMs or SMXs) as shown in Figure 5-1. Each SM or SMX is a SIMD processor with 8-32 CUDA cores called streaming processors (SPs). For example, Tesla C2050 has 14 SMs with total 448 SPs (i.e. 32 SPs per SM). Each SM has a fast *shared memory* (which is a type of on-chip local memory) and is shared by all

of its SPs. GPU also has a read-only *constant cache* and *texture cache* shared by all the SPs on the GPU (not shown in the figure). A set of local 32-bit or 64-bit *registers* is available for each SP. The SMs communicate through the *global/device* memory. The global memory can be read or written by the host, and is persistent across kernel launches by the same application. Shared memory is managed explicitly by the programmers [101].

5.2.2 CUDA Programming

CUDA (Compute Unified Device Architecture) developed by Nvidia was designed to efficiently support both graphical and non-graphical operations on GPU. CUDA's software interface provides ease of building and executing general applications on GPUs. The CUDA software stack is composed of a hardware driver, an application programming interface (API) and runtime library. CUDA applications are written in "C" and use the CUDA APIs to access the GPUs at runtime to get device information, copy data and load the computing workload to GPUs. The programming and execution model of CUDA applications usually involve the following steps:

1. Copy input data from main memory to GPU memory.
2. Launch kernel to be executed on GPU. Each kernel is a computational unit doing a specific task.
3. GPU executes the kernel that has been launched.
4. Copy output data back from GPU memory to main memory.

In CUDA, computation is written as a *kernel* on the CPU which is launched on GPU with a massive amount of similar computational units known as *threads* [99]. When a *kernel* is launched to execute on GPU, it is referred as a *grid of computation*. A *grid*

consists of multiple *thread blocks;* a block is a group of *threads*. A *thread block* is assigned to a multiprocessor SM. SM's execution model is SIMT (Single Instruction, Multiple Threads). Figure 5-2 depicts CUDA's execution model. *Threads* within the same *thread block* are divided into groups, called *warps*, each of which contains 32 threads. A *warp* of threads can combine accesses to consecutive data items in one device memory segment into a single memory access transaction called *coalesced access*. When a *grid* is executed on GPU, its *thread blocks* are distributed and assigned to execute on SMs; threads within a block are mapped into SPs of a single SM.

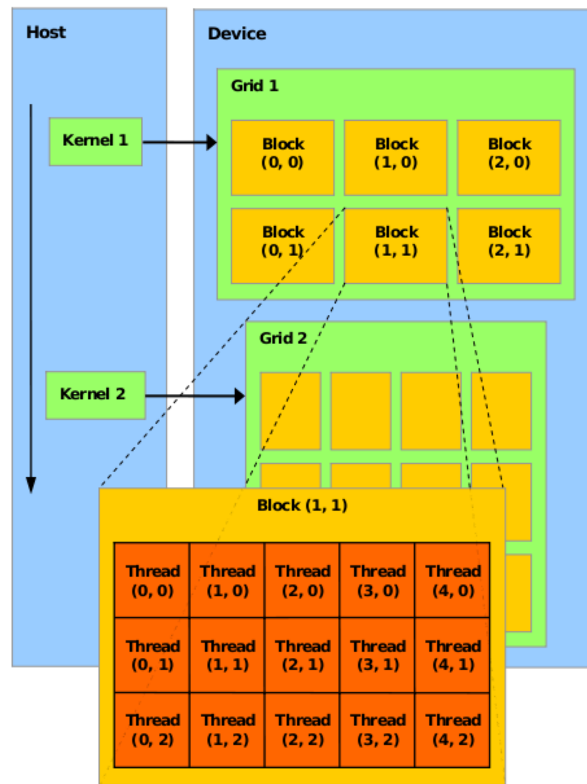


Figure 5-2: The execution model of CUDA

As a co-processor, GPU relies on the CPU for memory allocation. As such, the common practice for efficiency is to allocate the GPU memory statically before initiating the kernel and to avoid dynamic allocation or reallocation during kernel execution. Due

to the limited bus bandwidth between GPU and CPU memories, it is best to eliminate frequent, small-sized data transfers between CPU and GPU.

5.2.3 Frequent Pattern Mining Using GPU

Mining frequent patterns is nontrivial due to the necessity for amount of data and computational intensity over exponential search space. The GPU SIMT computational model (which for most practical purposes is very similar to SIMD: Single Instruction, Multiple Data [91]) works great for applications with massive amount of repetitive parallelism with regular access patterns. Performance penalties occur on GPUs due to irregular workloads, irregular access patterns, needs for synchronization among threads belonging to different blocks, and needs to move and access data in different levels of memory hierarchy. These pose significant challenges for FPM using GPU as computing resource.

The traditional FPM methods developed for sequential execution on CPU must be redesigned so that their computational model and data structures can exploit the type of parallelism provided by GPU architectures. GPU requires data representation that can be processed uniformly and independently by a large number of concurrent threads. In addition, data pre/post processing in CPU and transferring data between CPU and GPU can add a large enough overhead to the total execution time of FPM to negate the benefits of GPU.

5.3 Prior GPU-based FPM Algorithms

A review of sequential FPM algorithms is given in Section 2.3. Several GPU-based FPM methods have been developed. In this section, we research three most relevant GPU FPM algorithms CSFPM [103], GPApriori [106] and gpuDCI [104].

5.3.1 CSFPM Algorithm

Candidate Slicing Frequent Pattern Mining (CSFPM) [103] is one of Apriori-like methods for GPU. It off-loads the most time consume phase of counting to compute the *support*s to the GPU to speed up the total execution time. For better load balancing, the algorithm parallelizes and distributes the candidate *itemset*s to the GPU threads; each thread checks its own transaction in a candidate item. This reduces the processor waiting time since the load between processing units is more balanced.

5.3.2 GPApriori Algorithm

GPApriori by Zhang et al. [106] is also an Apriori-like method for GPU. It maps the Apriori algorithm to the SIMD execution model by using a "static bitset" memory structure to represent the input database. This data structure improves upon the traditional approach of the vertical data layout in state-of-the art Apriori implementations. Similar to CSFPM, GPApiori parallelizes only *support* counting step on the GPU while the remaining steps are executed on CPU. GPApriori applies several optimization techniques in its implementation: (1) before *support* counting is performed on GPU, candidates are preloaded to shared memory to prevent repeating global memory read, manual, (2) hand-tuned loop unrolling to further improve the kernel speed; and (3) hand-tuned block size [106].

5.3.3 gpuDCI Algorithm

gpuDCI [104] is an adoption of the DCI algorithm [71], a sequential mining approach that combines Apriori and Eclat. This algorithm starts its computation on CPU, as in DCI, and moves the pruned datasets to the GPU as soon as the bitwise vertical dataset fits into the GPU global memory. Afterwards the *support* computation is

performed on GPU. However, after switching to GPU, the CPU still manages patterns, generates candidates and stores patterns that are frequent according to the *support*s computed by the GPU. Two parallel techniques have been investigated: (1) for the *transaction-wise* technique, all GPU cores independent of the GPU multiprocessor they belong to, work on the same intersection or count operation; (2) for the *candidate-wise* technique, each GPU multiprocessor intersects and counts a different candidate. The *candidate-wise* technique has shown to perform better than the *transaction-wise* technique because it requires fewer synchronization operations.

5.4 New Frequent Pattern Mining Approach using a CPU-GPU Hybrid Model

5.4.1 The Proposed Multi-strategy Approach

Among many sequential frequent pattern mining methods that are traditionally developed for machines without GPU, FP-growth and its variants [26], [27], [28], [29], [30] are most efficient, especially for sparse large databases. They do not require generating a very large number of candidate itemsets as Apriori and Eclat do and hence save both memory and computation. The main mining computation of FP-growth is based on recursively generating FP-trees [25].

While the benefits of applying FP-growth can not be ignored, developing a method based on FP-growth for FPM poses a lot of challenges for GPU due to FP-tree data structure, recursive tree construction, and tree traversal need. GPUs perform best for tasks whose data structures are linear and computations lend themselves well to vector processing. Moreover, it has been shown that FP-growth does not perform as well as Eclat when mining dense databases or mining with low *minsup*s where the number of generated frequent pattern is very large [60], [32], [48]. For such cases, manipulating a

very large number of FP-trees in FP-growth becomes more costly than intersecting the TID-lists of Eclat. It is important to keep in mind that for TID-lists, the vertical and linear data formats and list intersection operations of Eclat are quite suitable for GPU but the depth-first approach does not allow creation of enough parallel workload, compared to Apiori, to fully utilize the large computing resources of GPU.

Therefore, we combine and redesign the advanced features of FP-growth, Eclat, MAFIA [10] and Apriori into a new mining method, CGMM, that applies both CPU and GPU computing to provide high FPM performance. As in DFEM, CGMM consists of two mining strategies and dynamically selects a suitable mining for each data portion of a database.

5.4.2 Overview of CGMM

CGMM consists of three main tasks: FP-tree construction, *CPUBasedMining* and *GPUBasedMining* described below. It starts with constructing the corresponding FP-tree followed by dynamically selecting between *CPUBasedMining* and *GPUBasedMining* similar to DFEM as depicted in Figure 5-3.

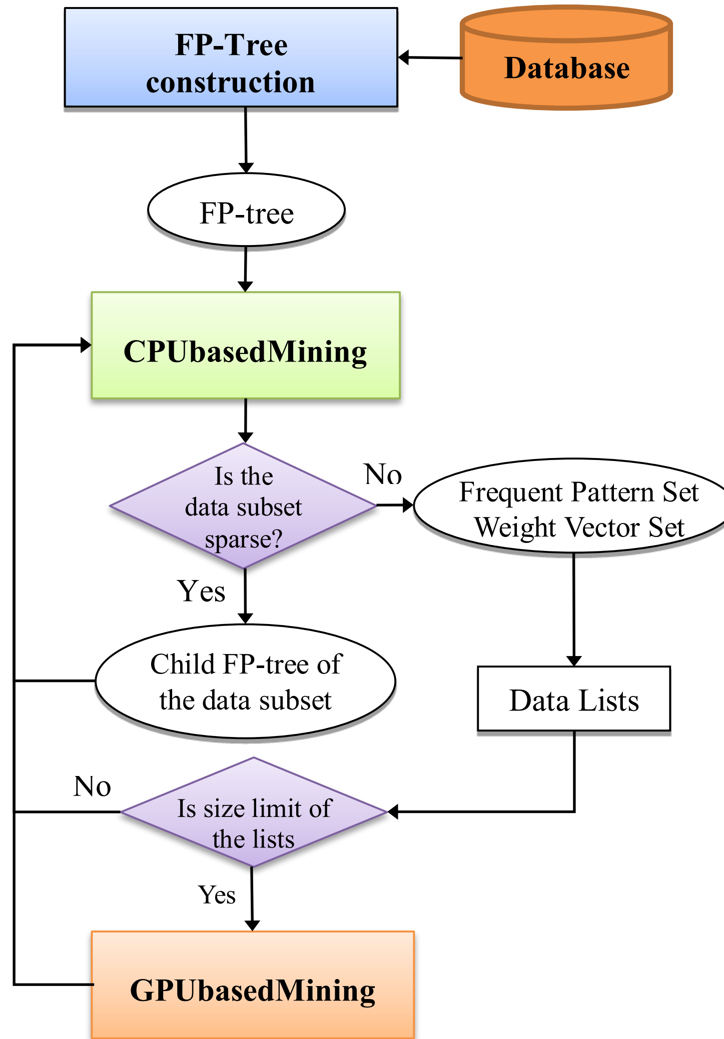


Figure 5-3: The overview of CGMM

FP-tree construction (on CPU) reads the database to build the corresponding FP-tree. Using this data structure significantly reduces the I/O cost because it compacts the database in memory before mining is performed. It also enables multiple mining strategies to be employed with relative ease as subsets of data from this tree can be used to create independent sub-mining tasks where each may use a different mining strategy.

CPUBasedMining (on CPU) extracts from the FP-tree the data subsets to recursively construct child FP-trees. The frequent patterns are identified based on newly

generated FP-trees without the requirement of generating a large number of frequent pattern candidates. Because FP-tree data structure is complex and inefficient for mining on GPU, only CPU is used to process the mining task. *CPUBasedMining* is distinct from the other mining methods because it is applied to sparse data subsets only. Determination of whether a data subset is sparse or dense is described in following section.

GPUBasedMining uses a new hybrid mining model designed for GPU applied to the dense data subsets. It presents data used to compute the *support* as bit vectors and maintains input and output data in data lists including *Frequent Pattern Set List* and *Weight Vector List*. The new frequent patterns are generated by applying candidate generation-and-test approach using a self-adaptive breath-first solution. Only a subset of frequent patterns is used to generate frequent pattern candidates at a time as long as their input and output data fit in the GPU memory. It addresses GPU memory limitation problem. Unlike the existing GPU solutions that off-load only the *support* counting phase to GPU, *GPUBasedMining* performs both the candidate generation and the *support* counting on GPU to increase GPU utilization and reduce overall processing on CPU as well as data transfer between CPU and GPU memories.

5.4.3 Switching Between the Two Mining Strategies

The multi-strategy approach of CGMM is enabled by the ability of switching between its two mining strategies *CPUBasedMining* and *GPUBasedMining*. It reapplies the switching condition for DFEM (Section 2.4.6). However, CGMM is different from DFEM, ShaFEM and SDFEM because the two mining strategies are executed on different computing devices, CPU and GPU. In Section 5.5, we present the data

structures used in CGMM and the impact of threshold K on CGMM performance and how to choose a good threshold in Section 5.6.

5.5 CGMM Algorithm

The CGMM algorithm is performed in two stages. The first stage is loading data into memory by constructing the FP-tree. Then, frequent patterns are generated using the two mining strategies in our algorithms by initially invoking *CPUBasedMining*, Figure 5-4.

CGMM algorithm
Input: Transactional database *D* and *minsup*
Output: Complete set of frequent patterns
1: Scan *D* once to identify all frequent items
2: Scan *D* a second time to construct the FP-tree *T*
3: Call **CPUBasedMining**($T, \varnothing, minsup$)

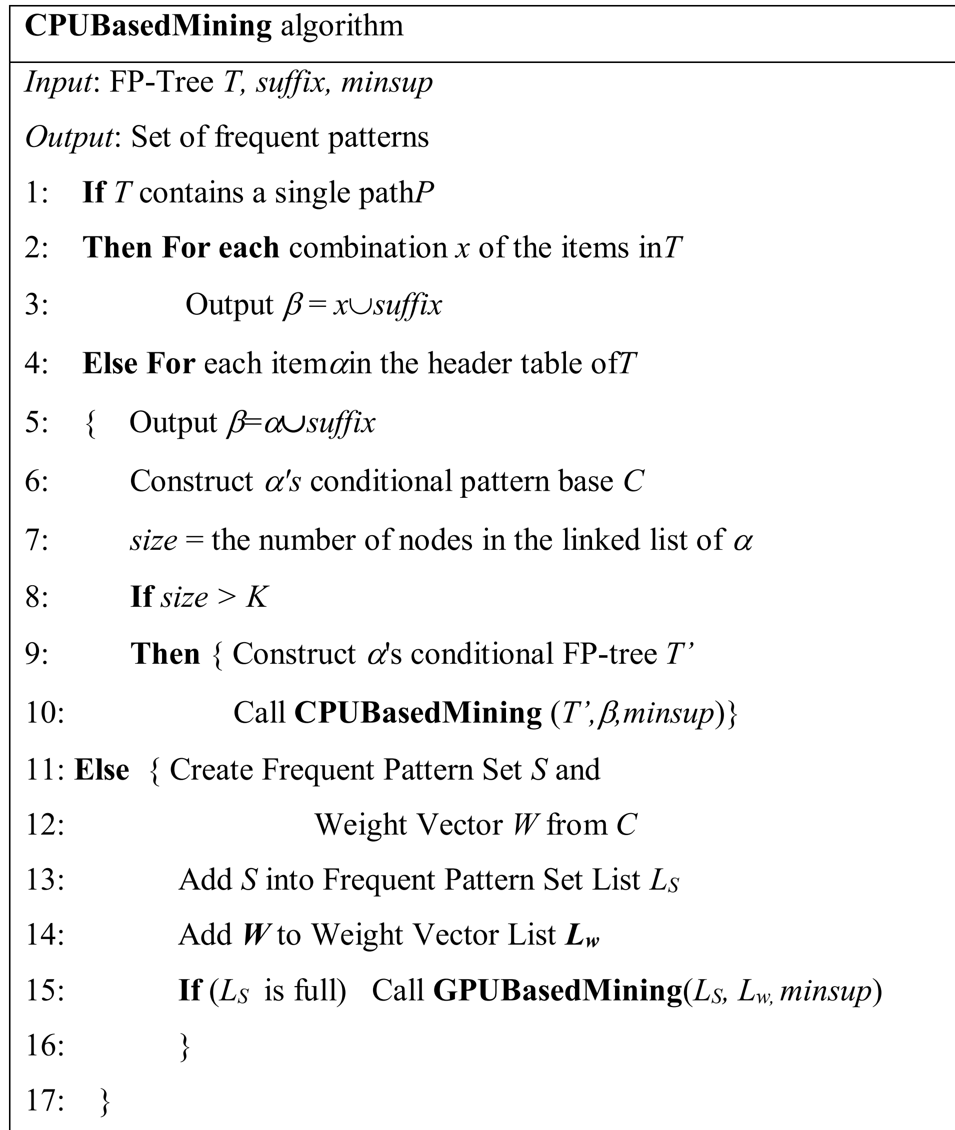
Figure 5-4: CGMM Algorithm

5.5.1 FP-tree construction and CPU Based Mining

The FP-tree construction phase is similar to that of FEM and DFEM, described in Section 2.4. *CPUBasedMining* is similar to *MineFPTree2* for DFEM, described in Section 2.6.2. For the FPTree mining approach, the switching stage is redesigned for GPU applications.

CPUBasedMining does not process data subsets which have dense characteristics. Instead, it converts them into *Frequent Pattern Set* and *Weight Vector* and adds them into the *Frequent Pattern Set List* and *Weight VectorList* managed by *GPUBasedMining*. As mining proceeds and when *CPUBasedMining* finds these lists full, it invokes

GPUBasedMining to start on GPU. The algorithmic description of *CPUBasedMining* is depicted in Figure 5-5.



CPUBasedMining algorithm

Input: FP-Tree *T, suffix, minsup*

Output: Set of frequent patterns

1: **If** *T* contains a single path*P*
2: **Then For each** combination *x* of the items in*T*
3: Output $\beta = x \cup suffix$
4: **Else For** each itemαin the header table of*T*
5: { Output $\beta = \alpha \cup suffix$
6: Construct α*'s* conditional pattern base *C*
7: *size* = the number of nodes in the linked list of α
8: **If** *size* > *K*
9: **Then** { Construct α's conditional FP-tree *T'*
10: Call **CPUBasedMining** (*T',*β*,minsup*)}
11: **Else** { Create Frequent Pattern Set *S* and
12: Weight Vector *W* from *C*
13: Add *S* into Frequent Pattern Set List L_S
14: Add ***W*** to Weight Vector List $\boldsymbol{L_w}$
15: **If** (L_S is full) Call **GPUBasedMining**(L_S, L_{w}, *minsup*)
16: }
17: }

Figure 5-5: CPUBasedMining algorithm

5.5.2 GPU Based Mining

GPUBasedMining mines the dense data portions of the database. It uses the GPU co-processor for most of its computational intensive needs, and CPU for the complicated tasks with data dependence to exploit the power and flexibility of GPU and CPU respectively.

5.5.2.1 Data structures

GPUBasedMining uses several data structures to manage the mining data including *Frequent Pattern Set*, *Frequent Pattern Set List* and *Weight Vector List* (Figure 5-6).

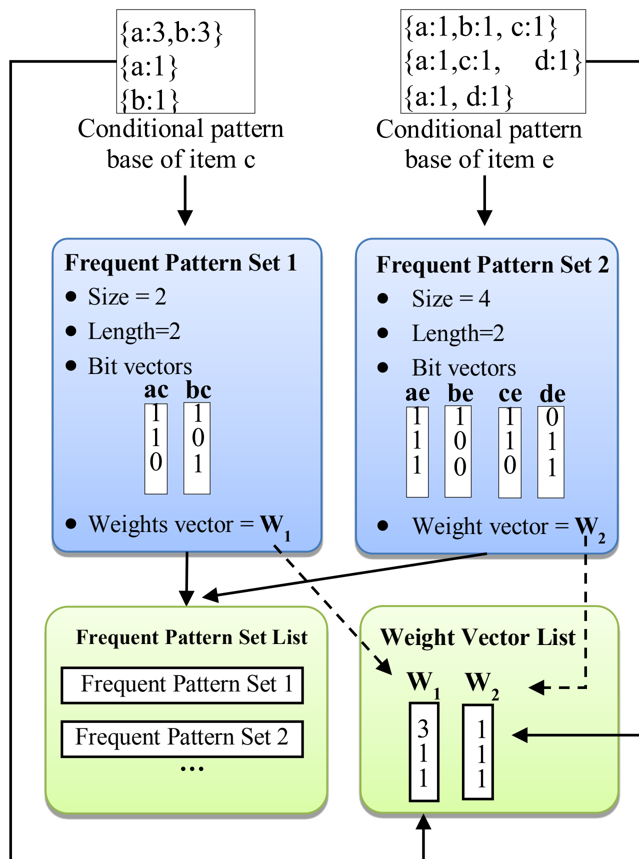


Figure 5-6: Data structures used by GPUBasedMining

Frequent Pattern Set is a set of frequent patterns that have same length k (i.e. they have *k* items in their itemsets) in which (k– 1) items are common and one item is different among the frequent patterns in the set. For example, three frequent patterns abc, abd, abe can form a *Frequent Pattern Set because they have ab in common*. It contains a

set of bit vectors where each presents the occurrence of a frequent pattern in the database. This data structure is used to generate new (k+1) length frequent pattern candidates and compute their *supports.* Some additional information of a *Frequent Pattern Set* includes s*ize* - the number of frequent patterns in the set, and *length* - the number of items in a frequent pattern. Figure 5-6 demonstrates two *Frequent Pattern Sets* that are created from the conditional pattern bases of items e and c extracted from the FP-tree. A *Frequent Pattern Set* is only created if it satisfies the condition for *GPUBasedMining* and contains the data of at least two frequent patterns.

Frequent Pattern Set List works as a container (buffer) that holds all *Frequent Pattern Sets* generated during the mining process and is updated by both CPUBasedMining and GPUBasedMining. In our example, two *Frequent Pattern Sets* are added into the *Frequent Pattern Set List.*

Weight Vector List: the weight vector is a portion of *Frequent Pattern Set* that is used to compute the *support* of the patterns and is created by collecting the frequency values of sets in the conditional pattern base. Because many *Frequent Pattern Sets* that are newly generated by *GPUBasedMining* may share a same weight vector, we store this data in a separate list called *Weight Vector List* and add a reference in *Frequent Pattern Set* to its *weight vector* in the list to avoid duplication and save memory.

5.5.2.2 Algorithmic description

During the execution of *CPUBasedMining*, small data subsets that meet the condition to be mined using *GPUBasedMining* are converted to *Frequent Pattern Sets* and added into the *Frequent Pattern Set List*. When the number of items in this list reaches a predefined limit, *GPUBasedMining* is invoked by *CPUBasedMining* to start

generating all new frequent patterns using *Frequent Pattern Set*s in the list. In Section 5.6, we present experiments showing the impact of different size limits on the performance of CGMM. The discovery of new frequent patterns from *Frequent Pattern Set List* involves the following steps; note that Steps 2 and 3, which comprise the most computational intensive phases of the program, are executed on GPU:

1. Extract a group of *Frequent Pattern Set*s from the list
2. Generate frequent pattern candidates using *Frequent Pattern Set*s
3. Compute the *support*s of candidates using *Frequent Pattern Set*s and *Weight Vector*s
4. Identify new frequent patterns from the candidates.
5. Add new *Frequent Pattern Sets* created from the new frequent patterns and repeat step 1 until no more *Frequent Pattern Set* is found from the list.

GPUBasedMining processes a group of *Frequent Pattern Set*s at a time by extracting multiple *Frequent Pattern Set*s from the list as long as their total memory size of newly generated candidates, their bit vectors and *Frequent Pattern Set*s used to generate them do not exceed the available memory on GPU. This workload computation helps CGMM flexibly scale to the memory size of physical device which is a major challenge in GPU computing. In addition, applying the data list structure allows *GPUBasedMining* to work without recursion (recursive procedures do not generally yield high performance on GPUs). Figure 5-7 presents the algorithmic description of *GPUBasedMining*. In this figure, the computation steps involving the GPU include lines 1, 6 – 9 and 13 respectively and are detailed in the following section.

GPUBasedMining algorithm
Input: Frequent Pattern Set List L_S ,Weight Vector List L_W, *minsup*
Output: Set of frequent patterns
1: M = Available memory on GPU
2: Copy L_W to the GPU
3: **While** L_S is not empty
4: { Specify $S \subset L_s$ where $m \leq M$
5: m = size of memory needed for S and candidates of S
6: Copy S to memory of GPU
7: Generate candidates on GPU using $\boldsymbol{S}$
8: Compute *support* of candidates on GPU using S
9: Copy *support* of candidates back to CPU
10: FP = new frequent patterns with $support \geq minsup$
11: Output FP
12: S_{new}= new *Frequent Pattern Set*s created using FP
13: Consolidate bit vectors of FP on GPU using S_{new}
14: Remove S from L_S
15: Add S_{new} to L_S
16: }

Figure 5-7: GPUBasedMining algorithm

5.5.2.3 Generating frequent pattern candidates and computing their Supports on GPU

The *Frequent Pattern Sets* and the *Weight Vector*s are copied to GPU and distributed among the *thread block*s. Each concurrent thread in a *thread bloc* will identify the candidates it needs to work on, using the information of *Frequent Pattern Sets*

assigned to its block, and determine necessary information of these candidates such as the two parent frequent patterns, their input bit vectors and its output bit vector to store the result of ANDing the parent'sbit vectors. Candidates are then generated in parallel by concurrent threads on GPU and their *support*s are computed. Each thread is responsible for the *support* of one or more candidates independently. Figure 5-8 illustrates the computations on GPU to generate the frequent pattern candidates and computation of the *support*s using the *Frequent Pattern Sets* and *Weight Vector*s in Figure 5-6.

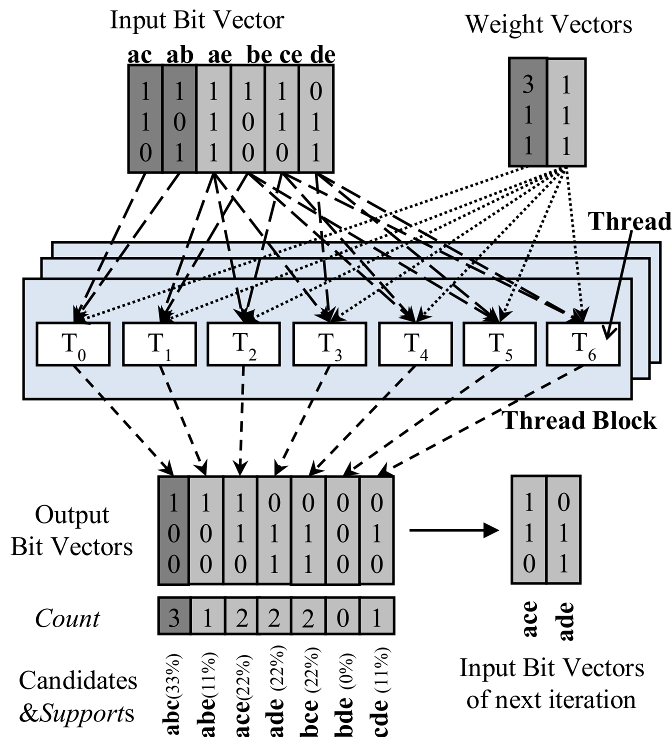


Figure 5-8: Generating pattern candidates and computing their counts on GPU

5.5.2.4 Data transfer optimization

An important feature of *GPUBasedMining* is that the output bit vectors of bitwise operations on GPU are not copied back to the main memory. Instead, the bit vectors belonging to the new frequent patterns are consolidated and used as the inputs in the next iteration. For this reason, the new *Frequent Pattern Sets* ($\mathbf{S}_{\mathbf{new}}$ in line 12 of Figure 5-7)

which are stored in main memory do not include bit vector data. This technique minimizes the communication cost between CPU and GPU, saves memory on the CPU side and enhances the overall performance of CGMM. For example, for *minsup*=20%, the new frequent patterns are abc, ace, ade, bce because their *support*s > 20%. Among those, abc, ade are used to create a new *Frequent Pattern Set* to add to the *Frequent Pattern Set List* because they can be used to create the candidates acde in the next iteration. Therefore, the bit vectors of abc, ade are kept and unified in GPU memory.

5.6 Performance Evaluation

5.6.1 Experimental Setup

Datasets: The six real datasets from Chapters 2, 3 and 4 are used as test cases. They represents various characteristics and domains of interest for our experiments: three sparse, one moderate and two dense databases all obtained from the FIMI Repository [59], a well-known repository for FPM. The database features are reported in Table 5-1.

Table 5-1: Experimental datasets of CGMM

Dataset	Type	# of Items	Average Length	# of Trans.
Chess	Dense	76	37	3196
Pumsb	Dense	2113	74	49046
Accidents	Moderate	468	33.8	340183
Retail	Sparse	16470	10.3	88126
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Software: CGMM can be implemented using different programming platforms like CUDA [97] or OpenCL [98]. In our experiments, we choose CUDA to implement CGMM because CUDA delivers better performance for the Nvidia GPUs used in our

experiments. We have carefully tested our implementation and have verified that it generates correct outputs in every case; this is often a challenge for complex applications developed on GPUs.

Hardware: We use an Altus 1702 machine with dual AMD Opteron 2427 processor, 2.2GHz, 24GB memory and 160 GB hard drive. This machine is equipped with NVIDIA Tesla Fermi C2050 GPU that has 3GB memory, 14 multiprocessors and each multiprocessor consists of 32 CUDA cores 1.5GHz. The operating system is CentOS 5.3, a Linux-based distribution.

5.6.2 Performance Evaluation

To evaluate performance of CGMM, we benchmarked it with six state-of-the-art FPM algorithms Apriori [8], Eclat [31], FP-growth [25], FP-growth* [26], AIM2 and DFEM (Section 2.6). Unlike CGMM with multi-strategy approach using both CPU (sequential execution) and GPU (parallel execution), these algorithms apply only one mining strategy and use CPU as the only computing engine. The running time of the seven methods on six datasets with various *minsups* are given in Figure 5-9. The experimental results show that CGMM outperforms the other algorithms including Apriori, Eclat, FP-growth, FP-growth* and AIM2 on both dense and sparse datasets for most test cases. Please note that the y-axis of the graphs is in logarithmic scale. CGMM does not run as well as DFEM for larger *minsup* values but it outperforms DFEM when *minsup* reduces. Hence, we recommend to apply CGMM for applications that uses low *minsup* values.

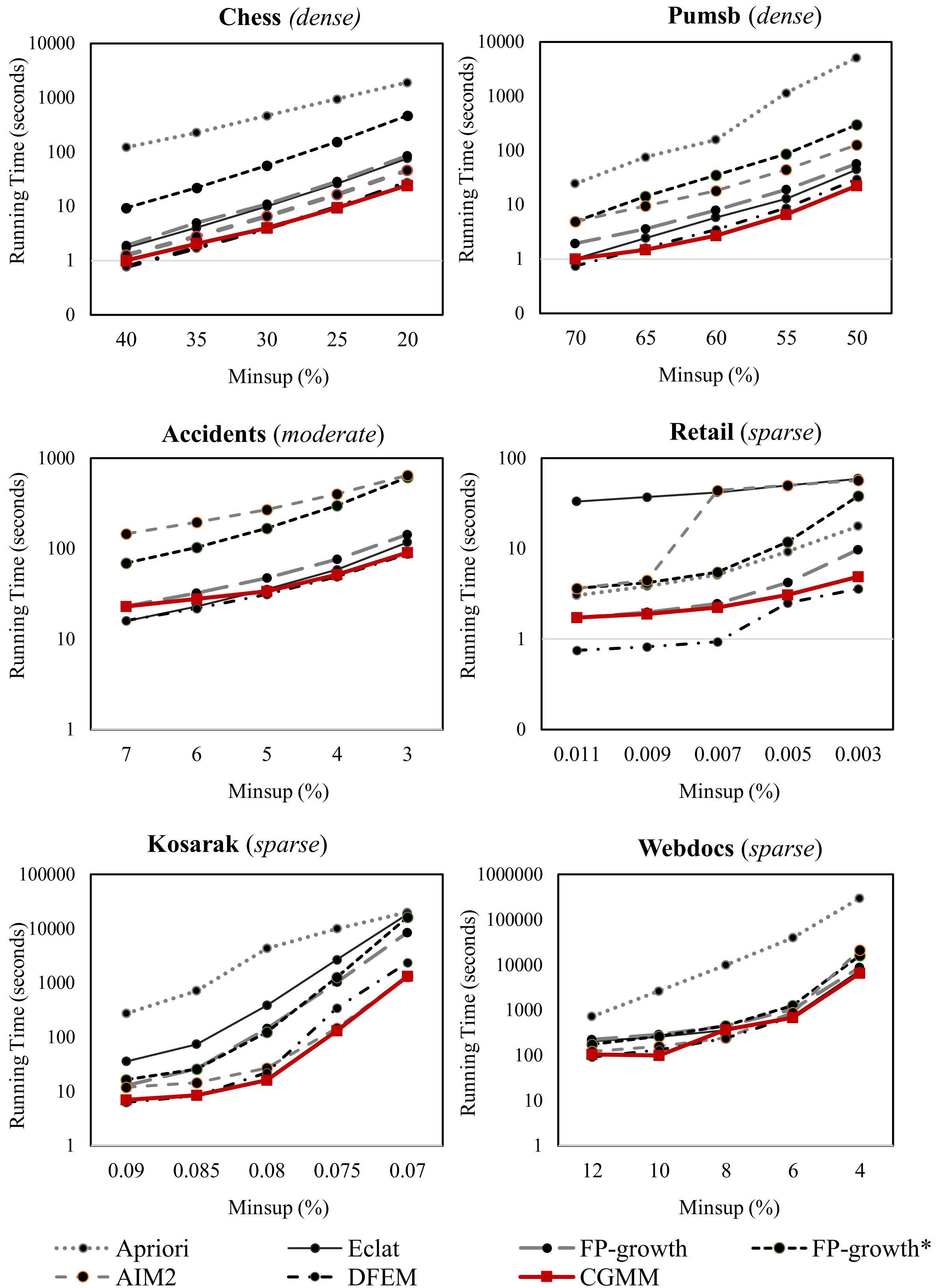


Figure 5-9: Running Time of CGMM vs. other sequential algorithms

For test cases with low *minsup* values, CGMM runs 1.0 – 229 times faster five compared algorithms except DFEM on test datasets (Table 5-2). It runs faster than DFEM 1.0 – 1.8 times on Chess, Pumsb, Accidents, Kosarak and Webdocs datasets. For Retail, DFEM performs better than CGMM. However, their time difference reduces as *minsup* is set to smaller values. When *minsup* is set to smaller values, the number of data subsets mined by *GPUBasedMining* of CGMM is large and GPU is more efficiently utilized. When *minsup* is larger, the amount of work delegated to GPU is small and this device is under-utilized. In such cases, the highly optimized DFEM is a better FPM solution.

Table 5-2: Speedup of CGMM vs. other sequential algorithms

Datasets	Minsup	vs. Apriori	vs. Eclat	vs. FP-growth	vs. FP-growth*	vs. AIM2	vs. DFEM
Chess	20%	78.4	3.1	3.5	19.2	1.9	1.1
Pumsb	50%	229.2	2.0	2.5	13.3	5.7	1.3
Accidents	3%	n/a	1.3	1.6	6.8	7.2	1.0
Retail	0.003%	3.7	12.2	2.0	7.8	11.7	0.7
Kosarak	0.08%	15.8	14.6	6.7	12.3	1.0	1.8
Webdocs	4%	45.5	1.1	1.3	2.4	3.3	1.1

We compares CGMM with GPApriori, a GPU based FPM algorithm (Section 5.3.2). Figure 5-10 shows CGMM runs 7.2 - 13.9 times faster than GPApriori on Retail dataset. In this test case, GPApriori uses GPU for entire dataset while CGMM uses GPU for only mining dense data subsets. GPApriori failed to run on other datasets because of the internal errors of this program. Upon the results on Retail, we find that for FPM problem, benefits of GPU is only obtained when we use it with suitable data structure and mining solution that can best leverage the computing power of GPU and adapt well to its limitation of memory and large data communication (e.g. mining dense data subset and low *minsup* values).

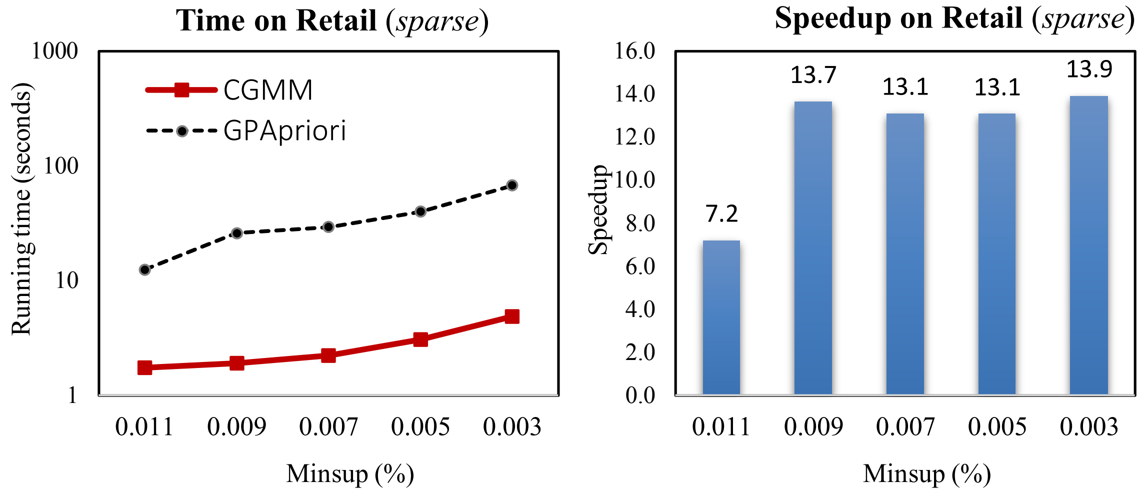


Figure 5-10: Running time and speedup of CGMM vs. GPApriori

5.6.3 Impact of Applying Multi-Strategy Approach

To study the benefits of applying the two mining strategies, we measured the time of CGMM in three separate cases: (1) using *CPUBasedMining* only, (2) using *GPUBasedMining*only and (3) using the combination of *CPUBasedMining* and *GPUBasedMining* as intended. The experimental results in Table 5-3 show that combining the two mining strategies significantly reduces the execution times for both sparse and dense databases compared to the cases where only one of the mining strategies were used. For example, CGMM with only *CPUBasedMining* took 1160 seconds to mine the Chess dataset while CGMM with both strategies ran in only 107 seconds which is 10.8 times faster. Similarly, for Accidents dataset, CGMM with both strategies performed 39.4 times faster than CGMM with only *GPUBasedMining* (i.e. 419 seconds vs. 16503 seconds). This performance gain comes from the ability to select the suitable strategy for each subset of data being mined to optimize the mining performance.

Table 5-3: Running time of CGMM with single mining strategy or both

Dataset	**CPUBasedMining** (seconds)	**GPUBasedMining** (seconds)	**CGMM** (seconds)
Chess	1160	180	**107**
Pumsb	2310	664	**377**
Accidents	547	16503	**419**
Retail	226	19	**18**
Kosarak	2773	841	**680**
Webdocs	6736	56017	**6496**

Additionally, we find that utilizing GPU for FPM does not always yield better performance. Although the computing throughput of GPU is hundred times larger than CPU. Use of GPU may sometimes downgrade the overall performance of a FPM task because of its memory limitations and data transfer overhead. That explains why for Accidents and Webdocs with very large memory requirements, *GPUBasedMining* performs much slower than *CPUBasedMining*. A combination of the two with a dynamic switching between them results in an improved overall performance.

5.6.4 Impact of Data Transfer Optimization

In designing GPU applications, attention to memory and data transfer requirements between CPU and GPU is a major part of algorithm design. In CGMM, grouping the output bit vectors of new frequent patterns on GPU memory and using them as input data in the next iteration, described in Section 5.5.2.3, results in significant performance improvement. As indicated in Table 5-4, this optimization step has a large impact in reducing cost of data transfer between the CPU and GPU memories. Otherwise, the output bit vectors of a large number of frequent pattern candidates are copied back to CPU memory which can later be copied back to GPU in order to compute the support in

the next iteration. In general, our optimization increases the overall performance between 10% - 210% for both sparse and dense databases in our experiments.

Table 5-4: Performance of CGMM with and without data transfer optimization

Dataset	**No Opt.** (1)(seconds)	**With Opt.** (2)(seconds)	**Time Difference** (3) = (1)-(2)(seconds)	**Speedup** (4) = (1) / (2)
Chess	176	107	69	1.6
Pumsb	687	377	310	1.8
Accidents	599	419	180	1.4
Retail	38	18	20	2.1
Kosarak	1029	680	349	1.5
Webdocs	6929	6496	433	1.1

5.6.5 Impact of Threshold K

Switching between two mining strategies of CGMM is based on a threshold K indicating the size of data subset as well as the data density characteristics. We study the impact of varying values of threshold K on the mining performance by measuring the execution time of CGMM with values of K ranging 128 – 4096. Results shown in Figure 5-11 indicate that for most cases CGMM runs faster as K increases (up to a point) because more dense data subsets meet the condition to mine using *GPUBasedMining*. For example, for Pumsb, CGMM runs at 1071 seconds for K = 128. Its execution time is reduced to 376-391 seconds when K ranges from 768 to 1024. However, when the value of K is increased to a certain point, especially for K≥1024 in our experiments, execution time of CGMM start to increase. This is caused by many data subsets with large sizes and sparse characteristics are selected to be mined with *GPUBasedMining* on GPU while *CPUBasedMining* is a better strategy for them. Although CGMM behaves differently for

different datasets, selecting a value of K within the range 640 – 1024 provides good performance for most datasets.

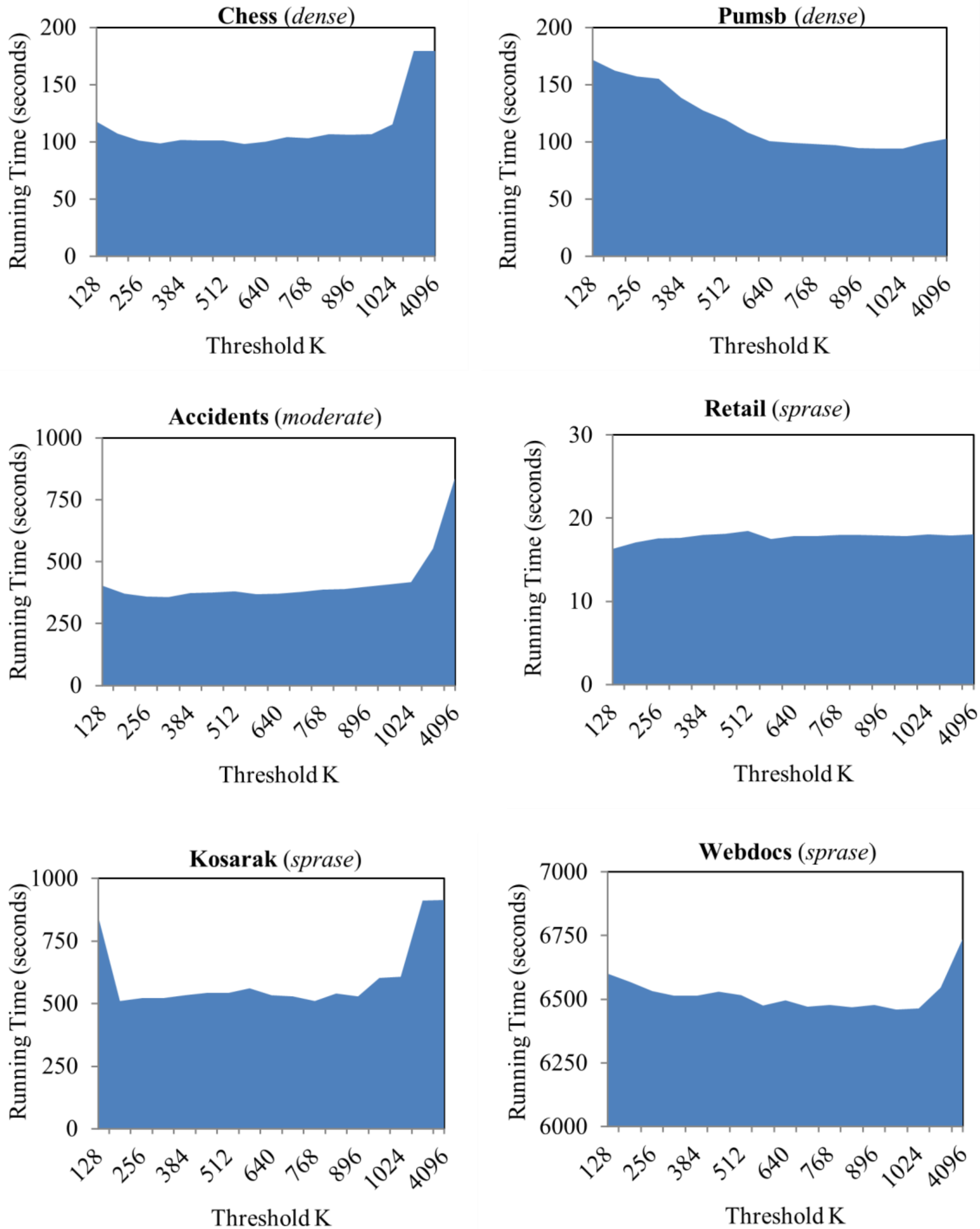


Figure 5-11 : Running time of CGMM with various values of threshold K

5.6.6 Impact of Data List Size

In *GPUBasedMining*, multiple *Frequent Pattern Set*s stored in *Frequent Pattern Set List* are processed as once so that GPU resources can be fully utilized. We tested CGMM with various maximum size of *Frequent Pattern Set List* to study its impact to the execution time of CGMM. Table 5-5 presents the experimental results. Note that the data list size indicates when *GPUBasedMining* can be invoked by *CPUBasedMining*.

Table 5-5: Execution time (sec.) of CGMM with different frequent pattern set list size

Dataset	**Size of Frequent Pattern Set List (LS)**				
	LS=1	*LS=10*	*LS=100*	*LS=1000*	*LS=10000*
Chess	112	110	108	107	107
Pumsb	394	380	377	364	344
Accidents	893	502	418	423	429
Retail	18	15	15	16	17
Kosarak	685	652	645	679	577
Webdocs	11582	8003	6738	6419	6427

From Table 5-5, we observe that the best performance for most test cases is when the size limit is between 1000 and 10000. Selecting a very large data list size results in more memory usage and larger GPU idle time, which in turn reduces the performance of CGMM. In contrast, selecting a small data list size, the amount of workload off-loading to GPU is not enough to fully utilize this device. We suggest to select a size limit that is larger than the number of computing units/cores in GPU for good performance of CGMM.

5.7 Conclusion

We have presented CGMM, a new CPU-GPU hybrid method for FPM. CGMM inherits FEM and uses GPU to mine dense data subsets of database. Its CPU based mining strategy sequentially mines sparse data similar to *MineFPTree* of FEM and its GPU based mining strategy deploys breath-first approach and bit vector data structure to parallel mine dense portions of database on GPU. Our experimental results show that CGMM runs up to 229 times faster than six sequential algorithms on six real datasets. Additionally, CGMM runs 7.2-13.9 times faster than GPApriori, a GPU based algorithm for FPM. Compared to DFEM, CGMM performs better DFEM in the test cases with low *minsup* values but worst in the other cases. We recommend to use CGMM as an alternative solution to DFEM for FPM applications that require low *minsup* input values.

6. Conclusion

Frequent Pattern Mining (FPM) has recently been used in many applications within different research domains, such as computational biology, computational biomedical imaging and business intelligence. The most common use of FPM in computational biology is discovery of frequent patterns of biological sub-sequences [109], [110], [111], gene expression correlations [17], [18], [20], [112], [113], [114] and functional annotations [19], [111], [113], [115], [116]. The use of FPM provides the foundation for effective solutions to the domain problems, namely motif finding, gene regulation prediction and gene/protein function prediction, without an essential requirement of a priori specifications of problem parameters, such as motif format, set of genes/experimental conditions of interest or group of functional annotations or limitations of analysis protocols. In Business Intelligence (BI), FPM is applied on transaction databases to search for frequent patterns of sold items, user's daily behaviour [21], [117], and tourism behaviour under various travel and living conditions [118]. Application of FPM in business data analysis allows providing customer-oriented information for promotion of a particular product, supporting business control decision making as well as fine-tuning business goals such as improving customer retention, or offering better personalized traveling experiences and services.

Due to the large amount of data generated in real-world applications, it is essential to have efficient high performance FPM methods for large-scale databases and for different computing platforms to support the data analysis and knowledge discovery from Big Data. In this dissertation, we have presented our newly developed and efficient FPM methods that address this demand and provide high performance for the FPM task on

different modern computer architectures such as multi-core multi-socket shared memory servers, multi-core clusters, and machines equipped with GPUs, etc. The research results include two new sequential FPM methods (FEM and DFEM) and three new parallel FPM methods (ShaFEM, SDFEM and CGMM) where each one is for a specific type of computers. We have shown the efficiency of these methods and analyzed the impact factor contributing the performance gain of our methods via many experiments presented in each chapter. Our research is unique and more advanced than existing methods for two main reasons:

- We develop a novel FPM approach that can self-adapt to the data based on their characteristics by applying multiple mining strategies and dynamic switching between them at runtime to provide best FPM performance on both sparse and dense databases. We integrate this mining approach in our five new FPM methods so that they can work efficiently on different data types. The novelty of detecting the characteristics of datasets as mining proceeds is a feature that can be applied with many mining strategies. Newly developed strategies can take advantage of this ability and adjust/adapt their mining using the most appropriate for the underlying dataset at runtime.
- We consider the advantages and disadvantages of different modern HPC computer architectures in designing our sequential and parallel FPM methods so that they can effectively leverage their computing power and memory resource. For example, we present optimization techniques to obtain best performance of FEM and DFEM (Chapter 2). Another example is applying shared memory programming model in ShaFEM and SDFEM to better utilize the benefits of

shared memory and to improve the FPM performance (Chapter 3 and 4). CGMM applies the breath–first approach for the GPU based mining strategy and the depth–first approach the CPU based one because the differences of hardware architecture makes each strategy be suitable for either GPU or CPU only (Chapter 5).

In future, we plan to extend our work on FPM to other hardware architectures like Intel's Xoen Phi co-processor or the multi-core cluster with GPUs. We will develop a MapReduce approach for FPM that bases on DFEM. Finally, we will investigate applying our newly developed FPM methods for Big Data applications.

REFERENCES

[1] "Top 10 Largest Databases in the World," in *http://www.comparebusinessproducts.com/fyi/10-largest-databases-in-the-world*, 2010.

[2] "You Tube's Statistic," in *https://www.youtube.com/yt/press/statistics.html*, 2014.

[3] "Google Search Statistics," in *http://www.internetlivestats.com/google-search-statistics/*, 2012.

[4] "How Big Is Facebook's Data? 2.5 Billion Pieces Of Content And 500+ Terabytes Ingested Every Day," in *http://techcrunch.com/2012/08/22/how-big-is-facebooks-data-2-5-billion-pieces-of-content-and-500-terabytes-ingested-every-day/*, 2012.

[5] J. Gantz and D. Reinsel, "The Digital Universe in 2020: Big data, Bigger Digital Shadow's, and Biggest Growth in the Far East," in *IDC*, December 2012.

[6] U. M. Fayyad, G. Piatetsky-Shapiro and P. Smyth, Advances in knowledge discovery and data mining, U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth and R. Uthurusamy, Eds., Menlo Park, CA, USA: American Association for Artificial Intelligence, 1996, pp. 1-34.

[7] J. Han, M. Kamber and J. Pei, Data Mining: Concepts and Techniques, 3rd ed., San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[8] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1994.

[9] J. Han, H. Cheng, D. Xin and X. Yan, "Frequent Pattern Mining: Current Status and Future Directions," *Data Mining and Knowledge Discovery,* vol. 15, no. 1, pp. 55-86, Aug 2007.

[10] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke and T. Yiu, "MAFIA: A Maximal Frequent Itemset Algorithm," *IEEE Transactions on Knowledge and Data Engineering,* vol. 17, no. 11, pp. 1490-1504, 2005.

[11] S. Brin, R. Motwani and C. Silverstein, "Beyond Market Baskets: Generalizing Association Rules to Correlations," in *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1997.

[12] C. Silverstein, S. Brin, R. Motwani and J. Ullman, "Scalable Techniques for Mining Causal Structures," *Data Mining and Knowledge Discovery,* vol. 4, no. 2-3, pp. 163-192, Jul 2000.

[13] R. Agrawal and R. Srikant, "Mining Sequential Patterns," in *Proceedings of the*

Eleventh International Conference on Data Engineering, Washington, DC, USA, 1995.

[14] H. Mannila, H. Toivonen and A. Inkeri Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery,* vol. 1, no. 3, pp. 259-289, jan 1997.

[15] J. Han, G. Dong and Y. Yin, "Efficient Mining of Partial Periodic Patterns in Time Series Database," in *Proceedings of the 15th International Conference on Data Engineering*, Washington, DC, USA, 1999.

[16] R. Agrawal, T. Imieliski and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of data*, New York, NY, USA, 1993.

[17] C. Becquet, S. Blachon, B. Jeudy, J.-F. Boulicaut and O. Gandrillon, "Strong-Association-rule Mining for Large-scale Gene-expression Data Analysis: A Case Study on Human SAGE Data," *Genome Biology,* vol. 3, no. 12, pp. research0067.1-research0067.16, 2002.

[18] E. Georgii, L. Richter, U. Rückert and S. Kramer, "Analyzing Microarray Data Using Quantitative Association Rules," *Bioinformatics,* vol. 21, no. suppl 2, pp. ii123--ii129, 2005.

[19] Y.-R. Cho and A. Zhang, "Predicting Protein Function by Frequent Functional Association Pattern Mining in Protein Interaction Networks," *IEEE Transactions on Information Technology in Biomedicine,* vol. 14, no. 1, pp. 30-36, 2010.

[20] M. J. Tarokha and E. Teymournejada, "Using Association Rules for Evaluation and Predicting Risk Factors in Business Intelligence System," *Business Intelligence Journal,* vol. 5, no. 2, 2012.

[21] W. Zhao, J. Liu, D. Ye and J. Wei, "Mining User Daily Behavior Patterns from Access Logs of Massive Software and Websites," in *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, 2013.

[22] G. Alaghband, "Parallel Architectures," in *Wiley Encyclopedia of Computer Science and Engineering*, John Wiley & Sons, Inc., 2007.

[23] M. J. Chorley and D. W. Walker, "Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-core Clusters," *Journal of Computational Science ,* vol. 1, no. 3, pp. 168-174, 2010.

[24] M. Sung, "SIMD Parallel Processing," *Architectures Anonymous,* vol. 6, p. 11, 2000.

[25] J. Han, J. Pei and Y. Yin, "Mining Frequent Patterns Without Candidate Generation," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2000.

[26] G. Grahne and J. Zhu, "Efficiently Using Prefix-trees in Mining Frequent Itemsets," in *Proceedings of the 2003 Workshop on Frequent Pattern Mining Implementations*, 2003.

[27] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang and D. Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases," in *Proceedings of the IEEE International Conference on Data Mining*, 2001.

[28] B. Racz, "nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree," in *Proceedings of the 2004 Workshop on Frequent Pattern Mining Implementations*, 2004.

[29] L. Liu, E. Li, Y. Zhang and Z. Tang, "Optimization of Frequent Itemset Mining on Multiple-Core Processor," in *Proceedings of the 33rd international conference on Very large data bases*, 2007.

[30] R. Moriwal, "FP-growth Tree for large and Dynamic Data Set and Improve Efficiency," *Information and Computing Science,* vol. 9, no. 2, 2014.

[31] 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*, 1997.

[32] S. Shporer, "AIM2: Improved Implementation of AIM," in *Proceedings of the 2004 Workshop on Frequent Itemset Mining Implementations*, 2004.

[33] H. Li, Y. Wang, D. Zhang, M. Zhang and E. Y. Chang, "PFP: Parallel FP-growth for Query Recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*, New York, NY, USA, 2008.

[34] M. Zaki, "Parallel and Distributed Association Mining: A Survey," *IEEE Concurrency,* vol. 7, no. 4, pp. 14-25, Oct. 1999.

[35] R. Garg and P. K. Mishra, "Some Observations of Sequential, Parallel and Distributed Association Rule Mining Algorithms," in *Proceedings of the 2009 International Conference on Computer and Automation Engineering*, Washington, DC, USA, 2009.

[36] H. D. K. Moonesinghe, M. Chung and P. Tan, "Fast Parallel Mining of Frequent Itemsets," in *Michigan State University*.

[37] S. K. Tanbeer, C. F. Ahmed and B.-S. Jeong, "Parallel and Distributed Frequent Pattern Mining in Large Databases," in *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, Washington, DC, USA, 2009.

[38] J. Li, Y. Liu, W. keng Liao and A. Choudhary, "Parallel Data Mining Algorithms for Association Rules and Clustering," in *CRC Press*, 2006.

[39] L. Vu and G. Alaghband, "A Fast Algorithm Combining FP-Tree and TID-List for Frequent Pattern Mining," in *Proceedings of the 2011 International Conference on Information and Knowledge Engineering*, 2011.

[40] L. Vu and G. Alaghband, "Mining Frequent Patterns Based on Data Characteristics," in *Proceedings of the 2012 International Conference on Information and Knowledge Engineering*, 2012.

[41] L. Vu and G. Alaghband, "An Efficient Approach for Mining Association Rules from Sparse and Dense Databases," in *Proceedings of the 2014 International Conference on Information and Knowledge Management, IEEE*, 2014.

[42] L. Vu and G. Alaghband, "Efficient Algorithms for Mining Frequent Patterns from Sparse and Dense Databases," *Intelligent Systems,* 2014.

[43] L. Vu and G. Alaghband, "Novel Parallel Method for Mining Frequent Patterns on Multi-core Shared Memory Systems," in *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems, ACM*, 2013.

[44] L. Vu and G. Alaghband, "Novel Parallel Method for Association Rule Mining on Multi-core Shared Memory Systems," *Parallel Computing*, 2014.

[45] L. Vu and G. Alaghband, "High Performance Frequent Pattern Mining on Multi-Core Cluster," in *Proceedings of the 2012 International Conference on Collaboration Technologies and Systems, IEEE*, 2012.

[46] L. Vu and G. Alaghband, "New Parallel Method for Frequent Pattern Mining on Multi-core Cluster," in *in submission*, 2014.

[47] L. Vu and G. Alaghband, "A Self-Adaptive Method for Mining Frequent Patterns using a CPU-GPU Hybrid Model," in *preparation for submission*, 2014.

[48] L. Schmidt-Thieme, " Algorithmic Features of Eclat," in *Proceedings of the 2004 Workshop on Frequent Itemset Mining Implementations*, 2004.

[49] W. Li and A. Mozes, "Computing Frequent Itemsets Inside Oracle 10g," in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, 2004.

[50] C. Utley, "Introduction to SQL Server 2005 Data Mining," in *Microsoft SQL server 9.0 technical articles*, 2005.

[51] T. Yoshizawa, I. Pramudiono and M. Kitsuregawa, "SQL Based Association Rule Mining using Commercial RDBMS (IBM DB2 UDB EEE)," in *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, 2000.

[52] M. Hahsler, B. Grun and K. Hornik, "arules - A Computational Environment for Mining Association Rules and Frequent Item Sets," *Journal of Statistical Software,*

vol. 14, no. 15, pp. 1-25, 9 2005.

[53] L. Hen and S. Lee, "Performance Analysis of Data Mining Tools Cumulating with a Proposed Data Mining Middleware," *Journal of Computer Science,* pp. 826-833, 2008.

[54] J. S. Park, M.-S. Chen and P. S. Yu, "An Effective Hash-Based Algorithm for Mining Association Rules," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of data*, New York, NY, USA, 1995.

[55] H. Toivonen, "Sampling Large Databases for Association Rules," in *Proceedings of the 22th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1996.

[56] S. Brin, R. Motwani, J. D. Ullman and S. Tsur, "Dynamic Itemset Counting and Implication Rules for Market Basket Data," in *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1997.

[57] N. Le, T. Nguyen and T. C. Chung, "BitApriori: An Apriori-Based Frequent Itemsets Mining Using Bit Streams," in *Procceedings of the 2010 International Conference on Information Science and Applications (ICISA)*, 2010.

[58] A. Ghanem and H. Sallam, "Hybrid Search Based Association Rule Mining," in *Procceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim)*, New York, NY, USA, 2011.

[59] "Frequent Itemset Mining Implementations Repository," in *Proceedings of the Workshop on Frequent Itemset Mining Implementation*, 2003-2004.

[60] A. Fiat and S. Shporer, "AIM: Another Itemset Miner," in *Proceedings of the 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.

[61] M. Zaki and K. Gouda, "Fast Vertical Mining Using Diffsets," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, New York, NY, USA, 2003.

[62] C. Borgelt, "An Implementation of the FP-growth Algorithm," in *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, New York, NY, USA, 2005.

[63] C. Borgelt, "Frequent Pattern Mining Implementations," *http://www.borgelt.net*.

[64] R. Rabenseifner, G. Hager and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Washington, DC, USA, 2009.

[65] OpenMP, *http://openmp.org*.

[66] K.-M. Yu, J. Zhou, T.-P. Hong and J.-L. Zhou, "A Load-balanced Distributed Parallel Mining Algorithm," *Expert Systems with Applications ,* vol. 37, no. 3, pp. 2459-2464, 2010.

[67] I. Pramudiono and M. Kitsuregawa, "Parallel FP-growth on PC Cluster," in *Proceedings of the 7th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, Berlin, Heidelberg, 2003.

[68] D. Chen, C. Lai, W. Hu, W. Chen, Y. Zhang and W. Zheng, "Tree Partition Based Parallel Frequent Pattern Mining on Shared Memory Systems," in *Proceedings of the 20th international conference on Parallel and distributed processing*, Washington, DC, USA, 2006.

[69] O. R. Zaiane, M. El-Hajj and P. Lu, "Fast Parallel Association Rule Mining without Candidacy Generation," in *Proceedings of the 2001 IEEE International Conference on Data Mining*, Washington, DC, USA, 2001.

[70] C. Bienia, S. Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, New York, NY, USA, 2008.

[71] S. Orlando, C. Lucchese, P. Palmerini, R. Perego and F. Silvestri, "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets," in *Proceedings of the 2003 Workshop on Frequent Itemset Mining Implementations*, 2003.

[72] T. Uno, M. Kiyomi and H. Arimura, "LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets," in *Proceedings of the 2004 Workshop on Frequent Itemset Mining Implementations*, 2004.

[73] AMD, "CodeAnalyst," *http://developer.amd.com/tools-and-sdks/archive/amd-codeanalyst-performance-analyzer*.

[74] Y. He and C. H. Ding, "MPI and OpenMP Paradigms on Cluster of SMP Architectures: The Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition," in *Supercomputing, ACM/IEEE 2002 Conference*, 2002.

[75] X. Wu and V. Taylor, "Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-scale Multicore Supercomputers," *SIGMETRICS Perform. Eval. Rev.,* vol. 38, no. 4, pp. 56-62, Mar. 2011.

[76] G. Hager, G. Jost and R. Rabenseifner, "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes," in *Proceedings of Cray User Group Conference*, 2009.

[77] M. El-hajj and O. R. Zaïane, "Parallel Leap: Large-scale Maximal Pattern Mining in a Distributed Environment," in *Proceeding of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)*, 2006.

[78] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan and C.-W. Tseng, "Dynamic Load Balancing of Unbalanced Computations Using Message Passing," *Parallel and Distributed Processing Symposium, International,* vol. 0, p. 391, 2007.

[79] K.-M. Yu, J. Zhou and W. Hsiao, "Load Balancing Approach Parallel Algorithm for Frequent Pattern Mining," in *Parallel Computing Technologies*, vol. 4671, V. Malyshkin, Ed., Springer Berlin Heidelberg, 2007, pp. 623-631.

[80] B. Manaskasemsak, N. Benjamas, A. Rungsawang, A. Surarerks and P. Uthayopas, "Parallel association rule mining based on FI-growth algorithm.," in *ICPADS*, 2007.

[81] B. Janaki Ramaiah, A. Rama Mohan Reddy and M. Kamala Kumari, "Parallel Privacy Preserving Association rule mining on pc Clusters," in *Proceedings of the IEEE International Advance Computing Conference (IACC 2009)*, 2009.

[82] K.-M. Yu and J. Zhou, "Parallel TID-based Frequent Pattern Mining Algorithm on a PC Cluster and Grid Computing System," *Expert Syst. Appl.,* vol. 37, no. 3, pp. 2486-2494, Mar. 2010.

[83] F. S. Tseng, Y.-H. Kuo and Y.-M. Huang, "Toward boosting distributed association rule mining by data de-clustering," *Information Sciences ,* vol. 180, no. 22, pp. 4263-4289, 2010.

[84] E. Ozkural, B. Ucar and C. Aykanat, "Parallel Frequent Item Set Mining with Selective Item Replication," *IEEE Transactions on Parallel and Distributed Systems,* vol. 22, no. 10, pp. 1632-1640, 2011.

[85] "Top 500," in *http://www.top500.org*.

[86] "IDC Market Study Shows Strong Gains for Co-Processors and Big Data at High Performance Computing Sites," in *http://www.idc.com/getdoc.jsp?containerId=prUS24176213*, 2013.

[87] "Message Passing Interface Forum, MPI: A Message-Passing Interface Standard," in *http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf*, 2012.

[88] Y. Sucahyo, R. Gopalan and A. Rudra, "Efficiently Mining Frequent Patterns from Dense Datasets Using a Cluster of Computers," in *AI 2003: Advances in Artificial Intelligence*, vol. 2903, T. Gedeon and L. Fung, Eds., Springer Berlin Heidelberg, 2003, pp. 233-244.

[89] D. Souliou, A. Pagourtzis, N. Drosinos and P. Tsanakas, "Computing frequent itemsets in parallel using partial support trees," *Journal of Systems and Software ,* vol. 79, no. 12, pp. 1735-1743, 2006.

[90] G. Goulbourne, F. Coenen and P. Leng, "Algorithms for computing association rules using a partial-support tree," *Knowledge-Based Systems ,* vol. 13, no. 2–3, pp. 141-149, 2000.

[91] L. E. Jordan and G. Alaghband, "Fundamentals of Parallel Processing," *Prentice Hall Professional Technical Reference*, 2002.

[92] "Open MPI: Open Source High Performance Computing," in *http://www.open-mpi.org/*.

[93] "MPICH," in *http://www.mpich.org/*.

[94] "LAM/MPI Parallel Computing," in *http://www.lam-mpi.org/*.

[95] T. Morgan, "Top 500 supers The Dawning of the GPUs," in *http://www.theregister.co.uk/2010/05/31/top_500_supers_jun2010/*, May 2010.

[96] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang, X. Gu and S. Zhang, "Cost effective data center servers," in *In the Proc. of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2013.

[97] Nvidia, "CUDA Programming," in *Best practices guide, http://www.nvidia.com/cuda*, 2013.

[98] A. Munshi, "OpenCL 1.0 Specification," in *Khronos OpenCL Working Group*, 2008.

[99] Q. Cui and X. Guo, "Research on Parallel Association Rules Mining on GPU," in *Proceedings of the 2nd International Conference on Green Communications and Networks 2012,* vol. 224, pp. 215-222, 2010.

[100] G. Teodoro, N. Mariano, W. M. Jr. and R. Ferreira, "Tree Projection-Based Frequent Itemset Mining on Multicore CPUs and GPUs," in *Proceedings of the Symposium on Computer Architecture and High Performance Computing,* vol. 0, pp. 47-54, 2010.

[101] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi and Y. Shi, "Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA)," *Supercomputing,* vol. 64, pp. 942-967, June 2013.

[102] Y. Kozawa, T. Amagasa and H. Kitagawa, "Parallel and Distributed Mining of Probabilistic Frequent Itemsets Using Multiple GPUs," *Lecture Notes in Computer Science, Database and Expert Systems Applications,* vol. 8055, pp. 145-152, 2013.

[103] C.-Y. Lin, K.-M. Yu, W. Ouyang and J. Zhou, "An OpenCL Candidate Slicing Frequent Pattern Mining algorithm on graphic processing units.," in *Procceedings of the 2011 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2011.

[104] C. Silvestri and S. Orlando, "gpuDCI: Exploiting GPUs in Frequent Itemset Mining," 2012.

[105] Y.-S. Huang, K.-M. Yu, L.-W. Zhou, C.-H. Hsu and S.-H. Liu, "Accelerating

Parallel Frequent Itemset Mining on Graphics Processors with Sorting," *Lecture Notes in Computer Science, Network and Parallel Computing,* vol. 8147, pp. 245-256, 2013.

[106] F. Zhang, Y. Zhang and J. Bakos, "GPApriori: GPU-Accelerated Frequent Itemset Mining," in *Procceedings of the 2011 IEEE International Conference on Cluster Computing*, Washington, DC, USA, 2011.

[107] Y. Kozawa, T. Amagasa and H. Kitagawa, "Fast Frequent Itemset Mining from Uncertain Databases using GPGPU," in *Procceedings of the Fifth International VLDB Workshop on Management of Uncertain Data*, 2011.

[108] J. Zhou, K.-M. Yu and B.-C. Wu, "Parallel Frequent Patterns Mining Algorithm on GPU," in *Procceedings of the 2010 IEEE International Conference on Systems Man and Cybernetics (SMC)*, 2010.

[109] W. Liu and L. Chen, "Efficiently Detecting Frequent Patterns in Biological Sequences," in *Web Information Systems and Applications Conference (WISA), 2011 Eighth*, 2011.

[110] K. M. Mutakabbir, S. S. Mahin and M. A. Hasan, "Mining Frequent Pattern Within a Genetic Sequence using Unique Pattern Indexing and Mapping Techniques," in *Proceedings of the 2014 International Conference on Informatics, Electronics & Vision (ICIEV)*, 2014.

[111] H. Motameni, H. A. Rokny and M. M. Pedram, "Using Sequential Pattern Mining in Discovery DNA Sequences Contain Gap," *American Journal of Scientific Research,* pp. 72-78.

[112] V. S. Tseng, H.-H. Yu and S.-C. Yang, "Efficient Mining of Multilevel Gene Association Rules from Microarray and Gene Ontology," *Information Systems Frontiers,* vol. 11, no. 4, pp. 433-447, 2009.

[113] R. Alves, D. S. Rodriguez-Baena and J. S. Aguilar-Ruiz, "Gene Association Analysis: A Survey of Frequent Pattern Mining from Gene Expression Data," *Briefings in bioinformatics,* vol. 11, no. 2, pp. 210-224, 2010.

[114] C. Creighton and S. Hanash, "Mining Gene Expression Databases for Association Rules," *Bioinformatics,* vol. 19, no. 1, pp. 79-86, 2003.

[115] Y.-H. Liu, "Mining Frequent Patterns from Univariate Uncertain Data," *Data \& Knowledge Engineering,* vol. 71, no. 1, pp. 47-68, 2012.

[116] P. Carmona-Saez, M. Chagoyen, F. Tirado, J. M. Carazo and A. Pascual-Montano, "GENECODIS: A Web-based Tool for Finding Significant Concurrent Annotations in Gene Lists," *Genome biology,* vol. 8, no. 1, p. R3, 2007.

[117] J. Song, T. Luo and S. Chen, "Behavior pattern mining: Apply Process Mining Technology to Common Event Logs of Information Systems," in *Proceedings of the IEEE International Conference on Networking, Sensing and Control (ICNSC*

2008), 2008.

[118] D.-s. Liu and S.-j. Fan, "Tourist Behavior Pattern Mining Model Based on Context," *Discrete Dynamics in Nature and Society,* vol. 2013, 2013.

[119] R. Rabenseifner, G. Hager and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Washington, DC, USA, 2009.

[120] A. Javed and A. Khokhar, "Frequent Pattern Mining on Message Passing Multiprocessor Systems," *Distributed and Parallel Databases,* vol. 16, no. 3, pp. 321-334, 2004.

[121] A. Don, E. Zheleva, M. Gregory, S. Tarkan, L. Auvil, T. Clement, B. Shneiderman and C. Plaisant, "Discovering Interesting Usage Patterns in Text Collections: Integrating Text Mining with Visualization," in *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, New York, NY, USA, 2007.

[122] J. Lifflander, S. Krishnamoorthy and L. V. Kale, "Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, New York, NY, USA, 2012.

[123] TMPGEnc, *http://tmpgenc.pegasys-inc.com*.

[124] Mathematica, *http://www.wolfram.com/mathematica*.

[125] badaboom, *http://www.badaboomit.com*.

[126] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn and T. Purcell, "A survey of general-purpose computation on graphics hardware," in *Computer Graphics Forum 07*, 2007.

[127] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo and P. Sander, "Relational Joins on Graphics Processors," in *Procceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2008.

[128] L. Sun, R. Cheng, D. W. Cheung and J. Cheng, "Mining Uncertain Data with Probabilistic Guarantees," in *Procceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2010.

[129] N. Govindaraju, J. Gray, R. Kumar and D. Manocha, "GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management," in *Procceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2006.

[130] L. Geng and H. J. Hamilton, "Interestingness Measures for Data Mining: A Survey," *ACM Computing Survey,* vol. 38, no. 3, Sep. 2006.

[131] C. Lucchese, "DCI Closed: A Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets," in *Procceedings of the IEEE ICDM 2004 Workshop on Frequent Itemset Mining Implementations (FIMI 2004)*, 2004.

[132] S.-M. Yu and S.-H. Wu, "An Efficient Load Balancing Multi-core Frequent Patterns Mining Algorithm," in *Proceedings of 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, 2011.

[133] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo, "Fast Discovery of Association Rules," *Advances in knowledge discovery and data mining,* pp. 307-328, 1996.

[134] S. Parthasarathy, M. Zaki, M. Ogihara and W. Li, "Parallel Data Mining for Association Rules on Shared Memory Systems," *Journal of Knowledge and Information Systems,* vol. 3, no. 1, pp. 1-29, Feb. 2001.

[135] R. Dass and A. Mahanti, "An Efficient Algorithm for Real-Time Frequent Pattern Mining for Real-Time Business Intelligence Analytics," in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, 2006.

[136] M. A. Upadhyay and M. B. Purswani, "Web Usage Mining has Pattern Discovery," *International Journal of Scientific and Research Publications,* vol. 3, no. 2, 2013.

[137] S. S. Jain, B. Meshram and M. Singh, "Voice of Customer Analysis Using Parallel Association Rule Mining," in *Proceedings of the 2012 IEEE Students' Conference on Electrical, Electronics and Computer Science (SCEECS)*, 2012.

[138] Y.-C. Liu, C.-P. Cheng and V. S. Tseng, "Discovering Relational-based Association Rules with Multiple Minimum Supports on Microarray Datasets," *Bioinformatics,* vol. 27, no. 22, pp. 3142-3148, 2011.

[139] T.-M. Chan, K.-C. Wong, K.-H. Lee, M.-H. Wong, C.-K. Lau, S. K.-W. Tsui and K.-S. Leung, "Discovering Approximate-associated Sequence Patterns for Protein--DNA Interactions," *Bioinformatics,* vol. 27, no. 4, pp. 471-478, 2011.

[140] H. R. H. Motameni and M. Pedram, "Using Sequential Pattern Mining in Discovery DNA Sequences Contain Gap," *American Journal of Scientific Research,* vol. 25, p. 72–78, 2011.

[141] R. V. Spriggs, Y. Murakami, H. Nakamura and S. Jones, "Protein Function Annotation from Sequence: Prediction of Residues Interacting with RNA," *Bioinformatics,* vol. 25, no. 12, pp. 1492-1497, 2009.

[142] C. Yu, N. Zavaljevski, V. Desai, S. Johnson, F. J. Stevens and J. Reifman, "The Development of PIPA: An Integrated and Automated Pipeline for Genome-wide

Protein Function Annotation," *BMC bioinformatics,* vol. 9, no. 1, p. 52, 2008.

[143] R. Martinez, N. Pasquier and C. Pasquier, "GenMiner: Mining Non-redundant Association Rules From Integrated Gene Expression Data and Annotations," *Bioinformatics,* vol. 24, no. 22, pp. 2643-2644, 2008.

[144] I. I. Artamonova, G. Frishman, M. S. Gelfand and D. Frishman, "Mining Sequence Annotation Databanks for Association Patterns," *Bioinformatics,* vol. 21, no. Suppl 3, pp. iii49--iii57, 2005.

[145] P. Weichbroth, M. Owoc and M. Pleszkun, "Web User Navigation Patterns Discovery from WWW Server Log Files," in *Proceedings of the 2012 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2012.

[146] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin and E. Chang, "Parallel Spectral Clustering in Distributed Systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 33, no. 3, pp. 568-586, March 2011.

[147] T. White, "Hadoop: The Definitive Guide: The Definitive Guide," *O'Reilly Media*, 2009.

[148] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communication. ACM,* vol. 51, no. 1, pp. 107-113, Jan. 2008.

[149] Z. Farzanyar and N. Cercone, "Efficient Mining of Frequent Itemsets in Social Network Data Based on MapReduce Framework," in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, New York, NY, USA, 2013.

[150] F. Kovacs and J. Illes, "Frequent Itemset Mining on Hadoop," in *Proceedings of the 2013 IEEE 9th International Conference on Computational Cybernetics (ICCC)*, 2013.

[151] Z. Zhang, G. Ji and M. Tang, "MREclat: An Algorithm for Parallel Mining Frequent Itemsets," in *Proceedings of the 2013 International Conference on Advanced Cloud and Big Data (CBD)*, 2013.

[152] H. Chen, T. Y. Lin, Z. Zhang and J. Zhong, "Parallel Mining Frequent Patterns Over Big Transactional Data in Extended Mapreduce," in *Proceedings of the Granular Computing (GrC)*, 2013.

[153] R. Ivancsy and I. Vajk, "Frequent Pattern Mining in Web Log Data," *Acta Polytechnica Hungarica,* vol. 3, no. 1, pp. 77-90, 2006.