Mining Frequent Sequences in One Database Scan Using Distributed Computers

by

Dale Allan Brajczuk

A thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada

August 2011

Thesis advisor

Dr. Carson K. Leung

Author

Dale Allan Brajczuk

Mining Frequent Sequences in One Database Scan Using Distributed Computers

Abstract

Existing frequent-sequence mining algorithms perform multiple scans of a database, or a structure that captures the database. In this M.Sc. thesis, I propose a frequent-sequence mining algorithm that mines each database row as it reads it, so that it can potentially complete mining in the time it takes to read the database once. I achieve this by having my algorithm enumerate all sub-sequences from each row as it reads it.

Since sub-sequence enumeration is a time-consuming process, I create a method to distribute the work over multiple computers, processors, and thread units, while balancing the load between all resources, and limiting the amount of communication so that my algorithm scales well in regards to the number of computers used. Experimental results show that my algorithm is effective, and can potentially complete the mining process in near the time it takes to perform one scan of the input database.

Table of Contents

Abstract ii
Table of Contents iv
List of Figures v
List of Tables viii
Acknowledgements ix
Dedication x

1 Introduction 1
1.1 Why Data Mining is Important 2
1.2 Frequent-Itemset Mining 5
1.3 Frequent-Sequence Mining 8
1.4 Frequent-Itemset and Frequent-Sequence Mining 9
1.5 Parallel and Distributed Computing 11
1.6 My Research 13

2 Related Work 17
2.1 Shared- and Distributed-Memory Computing Technologies 18
2.1.1 Hybrid MPI and OpenMP 18
2.1.2 Effectiveness of Multiple Thread Units 23
2.2 Load Balancing 24
2.2.1 Shared- and Distributed-Memory Computing Technologies Summary 28
2.3 Distributed Frequent-Pattern Mining Algorithms 29
2.3.1 Synchronous Partitioned Data Algorithms 30
2.3.2 Asynchronous Partitioned Data Algorithms 36
2.3.3 Asynchronous Duplicated Data Algorithms 40
2.3.4 Summary of Mining Approaches 47
2.4 An Ideal Framework for Mining from a Central Database 51
2.4.1 Requirements of the Ideal Algorithm 52
2.5 Summary 55

3 Single-Scan Sequential Mining Algorithm 58
3.1 Mining: Enumeration of Sub-Sequences . . . 60
3.1.1 Enumeration of Sub-Sequences with an Insertion List . . . 61
3.1.2 Depth-First Sub-Sequence Enumeration . . . 65
3.1.3 Sub-Sequence Enumeration on Shared Memory Computers . . 69
3.1.4 Summary on Mining or Enumeration of Sub-Sequences . . . 73
3.2 Result Collection . . . 75
3.2.1 Collection of Potentially Frequent Sequences . . . 75
3.2.2 Collection of All Sequences . . . 86
3.2.3 Summary of Results Collection . . . 91
3.3 Load Balancing . . . 92
3.4 Summary . . . 96

4 Experimental Evaluation 98
4.1 Hardware . . . 99
4.2 Datasets . . . 100
4.3 Experiments and Results . . . 101
4.3.1 Experiment: Number of Shared-Memory Processors . . . 102
4.3.2 Experiment: Number of Input Sequences . . . 113
4.3.3 Experiment: Input Sequence Length . . . 118
4.3.4 Experiment: Result Collection . . . 122
4.3.5 Experiment: Load Balancing . . . 126
4.3.6 Experiment: Number of Computers . . . 130
4.3.7 Experiment: Comparison with PrefixSpan . . . 137
4.4 Summary . . . 140

5 Conclusions and Future Work 145
5.1 Conclusions . . . 145
5.2 Future Work . . . 149

Bibliography 152

List of Figures

1.1 Example of a distributed memory system 14

2.1 Pure MPI model on a single computer 19
2.2 The Hybrid Master model during the communication stage 20
2.3 The Hybrid Master model during the computation stage 21
2.4 The Hybrid Overlap model 21
2.5 Theoretical runtime graphs 26
2.6 Stages of an SPDA; She et al.'s first distributed PrefixSpan 32
2.7 She et al.'s second distributed PrefixSpan 34
2.8 Stages of an APDA 37
2.9 Tanbeer et al.'s modified APDA 39
2.10 General ADDA structure 41
2.11 El-Hajj and Zaiane's FP-tree based ADDA 43
2.12 Cong et al.'s ADDA framework 46
2.13 Structure of the ideal algorithm 52

3.1 The sub-sequence enumeration of the sequence ⟨ a b c d ⟩ as a tree . 59
3.2 Enumerating sub-sequences with an insertion list 64
3.3 Depth-first sub-sequence enumeration 68
3.4 Tracing a node to parent path using an insertion list 78
3.5 Stages of my potentially frequent sequence collection algorithm 85
3.6 Stages of the collecting all sequences method 91

4.1 Runtime on an i7-920 using insertion mode 1 as the number of processes increases 103
4.2 Difference between runtime and predicted time on an i7-920 using insertion mode 1 as the number of processes increases 105
4.3 The memory usage of insertion mode 1 on an i7-920 using insertion mode 1 as the number of processes increases 105
4.4 Runtime on an i7-920 using insertion mode 2 as the number of processes increases 106

4.5 Difference between runtime and predicted time on an i7-920 using insertion mode 2 as the number of processes increases 106
4.6 The memory usage of insertion mode 1 on an i7-920 using insertion mode 2 as the number of processes increases 107
4.7 Comparison of insertion mode 1 and 2 runtimes on an i7-920 as the number of processes increases 108
4.8 Comparison of insertion mode 1 and 2 memory on and i7-920 as the number of processes increases 108
4.9 Runtime on a Q6600 using insertion mode 1 as the number of processes increases 109
4.10 Difference between runtime and predicted time on a Q6600 using insertion mode 1 as the number of processes increases 110
4.11 Runtime on a Q6600 using insertion mode 2 as the number of processes increases 111
4.12 Difference between runtime and predicted time on a Q6600 using insertion mode 2 as the number of processes increases 111
4.13 Comparison of insertion mode 1 and 2 runtimes on a Q6600 as the number of processes increases 112
4.14 Runtime of insertion mode 1 as the number of input rows increases . 115
4.15 Memory use of insertion mode 1 as the number of input rows increases 115
4.16 Runtime of insertion mode 2 as the number of input rows increases . 116
4.17 Memory use of insertion mode 2 as the number of input rows increases 116
4.18 Comparison of the runtimes of insertion modes 1 and 2 as the number of input rows increases 117
4.19 Comparison of the memory requirements of insertion modes 1 and 2 as the number of input rows increases 117
4.20 Runtime of insertion mode 1 as the input row length increases 119
4.21 Runtime of insertion mode 2 as the input row length increases 119
4.22 Memory use of insertion mode 1 as the input row length increases . . 120
4.23 Memory use of insertion mode 2 as the input row length increases . . 120
4.24 Comparison of runtimes of insertion modes 1 and 2 as the input row length increases 121
4.25 Comparison of the memory requirements of insertion modes 1 and 2 as the input row length increases 121
4.26 Runtime of collection mode 1 as the minimum support decreases . . . 124
4.27 Number of nodes collection mode 1 collects as the minimum support decreases 124
4.28 Component runtimes of insertion mode 2 as the minimum support decreases 125
4.29 Comparison of collection mode 1 and 2 runtimes as the minimum support decreases 126
4.30 Runtime comparison between heterogeneous computers 128

4.31 The percentage change of runtimes between input datasets 128
4.32 Portion sizes based on three input datasets 129
4.33 Component runtimes of collection mode 1 as the number of computers increases 131
4.34 Component runtimes of collection mode 1 as the number of computers increases 131
4.35 The total runtime of collection mode 1 as the number of computers increases 133
4.36 Component runtimes of collection mode 2 as the number of computers increases 134
4.37 Detailed view of the saving and loading times required by collection mode 2 as the number of computers increases 135
4.38 Comparison of the average runtimes of collection modes 1 and 2 as the number of computers increases 136
4.39 The runtime of PrefixSpan in comparison to the runtime of my algorithm using four heterogeneous computers, one computer with multiple processes, and one computer while limited to one process 137
4.40 The memory usage of PrefixSpan in comparison to the runtime of my algorithm using four heterogeneous computers, one computer with multiple processes, and one computer while limited to one process 138
4.41 The runtime of PrefixSpan as the minimum support threshold decreases, compared to my algorithm 139
4.42 The memory usage of PrefixSpan as the minimum support threshold decreased, compared to my algorithm 140

List of Tables

1.1 Grocery store example 5
1.2 Grocery store example: Frequent 2-itemsets and candidate 3-itemsets 6
1.3 Grocery store example: All frequent-itemsets 6
1.4 Frequent-sequence mining example 8
1.5 Frequent-sequence mining example: Frequent 2-sequences and potentially frequent 3-sequences 9

2.1 Example sequential database 29
2.2 Database with projection offsets 30
2.3 *a*-projected database 31
2.4 A database partitioned evenly between two computers 38

3.1 The sub-sequence enumeration of the sequence ⟨ a b c d ⟩ 59
3.2 Sizing partitions based on benchmark database runtimes 95

4.1 Structure of an input data file 100

Acknowledgements

Foremost, I thank Dr. Carson Leung, for the guidance and support he has provided throughout my research, as well as giving me the opportunity to participate in other research projects.

I also thank the Department of Computer Science Graduate Studies Committee for their feedback on my thesis proposal, which I used to guide my thesis write-up. Extra thanks go to Dr. Eskicioglu, for taking the time to discuss the Committee's concerns, and giving me suggestions of how to alleviate them.

I would also like to thank the members of my thesis examination committee, Dr. Athula Rajapakse, Dr. Pourang Irani, and the chair of my thesis defence, Dr. Rasit Eskicioglu.

To all of my family and friends, thank you for supporting me through this process.

Dale Allan Brajczuk
B.C.Sc.(Honours), University of Manitoba, Canada, 2006

The University of Manitoba
August 2011

To Amber, for being supportive. And patient.

Chapter 1

Introduction

Data mining—the search for implicit, previously unknown, and potentially useful information from large collections of data—has been an important topic in computer science since its introduction. Researchers have created multiple methods of discovering this information, and have divided data mining research into these sub-topics, the most important of which are *association rule mining*, *clustering*, and *classification*. Association rule mining, which is the focus of my research, is the search for correlations between items in a database. For example, a grocery store manager may be interested in how often different items are purchased together to plan sales and price increases, or a hospital administrator may want to find the recovery rates for a disease after a sequence of treatments to determine the best course of action. In these examples, the grocery store manager uses frequent-itemset mining, where the order of items in each purchase does not matter, and the hospital administrator uses frequent-sequence mining, where a different ordering of treatments results in a different recovery rate.

Clustering and classification are methods of organizing data. Clustering algorithms require a method to measure how similar items are to each other. They use this similarity measurement to assign each item in the database to a subset, called a cluster, such that all items within each cluster are similar to each other, and dissimilar to items in other clusters. Classification uses a subset of the entire dataset, called training data, to build a classification model so that it can classify new data, the test data, into the same groups. Clustering techniques may be used for obtaining the training data.

I have focused my research on association rule mining, specifically frequent-sequence mining, using multiple computers. To introduce my research, I have first shown that data mining is an important topic in computer science, by showing some of the projects it has made possible.

1.1 Why Data Mining is Important

Researchers have applied data mining techniques, such as frequent-sequence mining, to data collected in many fields. One of the most active of these is cancer research. Lisboa et al. [LVT$^+$10] stated, in their overview of data mining in cancer research, that applying data mining techniques to genetic sequences and patient histories discovers correlations leading to the identification of disease sub-types and the likely outcome of prognoses. Mining genetic sequences to discover correlations with different types of cancer, such as Giarratana et al.'s research on breast cancer [GPM$^+$09] and Wu et al.'s research on prostate cancer [WFC09], starts by determining how similar genetic sequences are to each other. Their algorithms cluster similar genes into discreet categories, and then use frequent-sequence mining to discover how frequently each gene category results in cancer. They readjust the similarity measurement in their clustering method until frequent-sequence mining gives a strong correlation.

Performing frequent-sequence mining on patient histories, including risk factors identified by clustering, treatments, and outcomes, discovers information such as what a sequence of risk factors leaves a patient at risk for, what prognosis a sequence of symptoms leads to, and which sequence of treatments is most effective. Yuguang et al. [YCM11] applied frequent-sequence mining to hospital databases to discover which risk factors frequently lead to diseases, such as frequently diagnosing smokers between age 40 and 50 with chronic bronchitis, so that hospitals can start pre-emptive treatment. Shamin et al. [SSM10] described the complete process of mining useful information from medical databases, and emphasize the fact that multiple data mining methods are required. They point out that while many studies, such as the previously mentioned works by Giarratana et al. and Shamin et al., identify genes correlated with specific diseases, most diseases are caused by combinations of genes working together. The authors use frequent-itemset and frequent-sequence mining to discover groups of genes or proteins, and study their interactions and relationships.

Song et al. [SLC08] have applied data mining techniques to system event logs, specifically, event logs from social networking sites. By creating a data pre-processing system specific to social networks, and using ideas from stream mining algorithms, the authors created a system capable of discovering frequent user behaviours. They use these frequent behaviours to create a model that predicts the behaviour of future users, and identify areas of social networking sites that would benefit most from optimization. Similarly, Xiongyan et al. [LLW09] applied data mining techniques to model aggregate data from geological databases, physical properties databases, seismic database, and logged data from oil and water exploration. The resulting model predicts where oil reservoirs are located, and directs future exploration.

Thuraisingham and Khan [Thu09] used various data mining techniques in the area of malicious code detection and security applications. They used anomaly detection techniques to identify behaviour patterns differing from the norm, along with analysis of frequent patterns leading up to criminal activity, to identify behaviours that are likely leading to criminal activity. They also used these methods to detect malicious code by discovering similarities to known malicious code, detect network intrusions by finding common traffic patterns associated with attacks, and analyze audit logs for signs of tampering or other undesired activity.

Paranjape-Voditel and Deshpande [PD11] used frequent-sequence mining to perform stock market predictions. Where most stock-prediction algorithms analyze the history of a specific stock, using frequent-sequence mining, the authors can identify relationships between multiple different stocks. For instance, if the stock of a company increases due to the popularity of their new product, the stocks of companies that produce components of the product may also increase.

Young et al. [YFP$^+$10] studied how data mining techniques can improve the operational availability of aircraft launch and recovery equipment by influencing maintenance schedules. The authors discovered which actions are most likely to resolve specific problems by mining previous maintenance logs to see which problems, changes, and resolutions occur together frequently. This reduces the amount of time spent

Table 1.1: Grocery store example

Transaction ID	Items Purchased
t_1	{ bread, bacon, milk, eggs, apples }
t_2	{ broccoli, carrots, chicken, zucchini }
t_3	{ eggs, bacon, oranges, buns }
t_4	{ tomatoes, lettuce, bacon, buns }
t_5	{ cereal, milk, bacon, eggs }

testing and replacing components. Their research also discovers patterns leading to component failure, which allows them to predict these failures, and perform maintenance before the failure occurs. Further, it identifies checks that mechanics perform too frequently or unnecessarily, reducing the amount of unneeded maintenance, and increasing the overall operational ability.

1.2 Frequent-Itemset Mining

Agrawal et al. [AIS93] first drew attention to data mining in 1993 when they presented the Apriori algorithm for *frequent-itemset mining.* A *frequent-itemset* is a collection of items that occur together as or more often than a user-defined threshold, called the *minimum support threshold*, in a database or dataset. The user sets the minimum support threshold such that those itemsets occurring less frequently than the threshold dictates are not of interest. The Apriori algorithm prunes uninteresting itemsets as it searches for frequent-itemsets so that it does not need to spend time processing them.

An example of the Apriori algorithm clarifies the related definitions, as well as the goal of data mining in general. Table 1.1 represents the items purchased from a

Table 1.2: Grocery store example: Frequent 2-itemsets and candidate 3-itemsets

Candidate 2-Itemsets	Count	Candidate 3-Itemsets
{ bacon, buns }	2	{ bacon, eggs, milk }
{ bacon, eggs }	3	
{ bacon, milk }	2	
~~{ buns, eggs }~~	~~1~~	
~~{ buns, milk }~~	~~0~~	
{ eggs, milk }	2	

Table 1.3: Grocery store example: All frequent-itemsets

Itemset	Count
bacon	4
buns	2
eggs	3
milk	2
{ bacon, buns }	2
{ bacon, eggs }	3
{ bacon, milk }	2
{ eggs, milk }	2
{ bacon, eggs, milk }	2

grocery store. Each row in the table is a sales transaction, which is composed of a unique transaction id, and all of the items purchased in that transaction. For this example, the minimum support threshold, shortened to *minSup*, is 40%. This means that an itemset must occur in two or more transactions to be frequent, since 40% of the 5 total transactions is 2.

Since frequent-itemsets are mathematical sets, as opposed to sequences, the Apriori frequent-itemset mining algorithm is not concerned with the order in which items appear in the row. In the case of the grocery store example, it is not concerned with the order items are wrung up in, only that they are all purchased together. The Apriori algorithm takes advantage of this by sorting the items within each transaction into lexicographical order. This sorting means that there are not multiple permutations of each set for the algorithm to handle. The algorithm performs multiple scans of the database, with the first scan sorting the items. In the k^{th} database scan, the algorithm counts all itemsets of length k and uses the frequent itemsets to create a list of itemsets of length $k+1$ that are potentially frequent, called candidate itemsets, to count on the next database scan. The algorithm creates candidate itemsets by first overlapping the last $k-1$ items from one frequent itemset with the first $k-1$ items from another, and then ensuring that all subsets of length k of the resulting set are also frequent.

Table 1.2 shows the candidate 2-itemsets computed on the first scan, their actual counts from the second scan and the resulting frequent 2-itemsets, and the candidate 3-itemsets that the algorithm will count on the third scan. Table 1.3 shows all of the frequent itemsets in the example database. Data miners use these frequent itemsets to discover correlations between items. This is new knowledge discovered from the database, and is ultimately the goal of data mining. They show the strength of correlations by dividing the number of times a frequent-itemset occurs by the number of times a subset of it occurs. In this example, customers purchased bacon 4 times, eggs 3 times, and bacon and eggs together 3 times. This gives the correlations that 75% of customers who bought bacon also bought eggs ($\{bacon, eggs\} : 3/bacon : 4$), and 100% of customers who bought eggs also bought bacon ($\{bacon, eggs\} : 3/eggs : 3$).

Table 1.4: Frequent-sequence mining example

User ID	Items
1	$\langle$ a b c $\rangle$
2	$\langle$ a c b $\rangle$

1.3 Frequent-Sequence Mining

While frequent-itemset mining reveals correlations between frequently occurring items, there are situations in which finding temporal correlations between these frequently occurring items is important. Agrawal and Srikant [AS95] recognised the need to find *frequent-sequences* and presented their AprioriAll and AprioriSome frequent-sequence mining algorithms in 1995. A frequent-sequence is a series of items that occur in a database, in the same order, as or more often than the minimum support threshold. The difference between the AprioriAll and AprioriSome algorithms is that AprioriSome only returns *maximal* frequent-sequences, that is, sequences that have no frequent superset.

The difference between the frequent itemset-mining Apriori algorithm and the frequent-sequence mining algorithms, AprioriAll and AprioriSome, is that the frequent-sequence mining algorithms cannot sort the items within each transaction into lexicographical order. This changes the number of candidate sequences that the algorithms must test in the candidate generation stage. Where the Apriori algorithm has a maximum of $(n-1)n/2$ candidates from n frequent-itemsets, AprioriAll and AprioriSome have twice as many maximum candidates from n frequent-sequences, $(n-1)n$, since each frequent-sequence can appear before or after each other frequent-sequence.

Consider the two rows in Table 1.4, representing the web pages a user requests

Table 1.5: Frequent-sequence mining example: Frequent 2-sequences and potentially frequent 3-sequences

Candidate 2-sequences	Count	Candidate 3-sequences
$\langle$ a b $\rangle$	2	
$\langle$ a c $\rangle$	2	
~~$\langle$ b a $\rangle$~~	~~0~~	
~~$\langle$ b c $\rangle$~~	~~1~~	
~~$\langle$ c a $\rangle$~~	~~0~~	
~~$\langle$ c b $\rangle$~~	~~1~~	

from a website, with a minimum support threshold of 2. Each of the pages, a, b, and c, are frequently requested since all three are viewed by both users. Users can potentially visit each page before or after each other page, resulting in the candidate 2-sequences shown in Table 1.5. Using the same candidate generation criteria as the Apriori algorithm for frequent-itemset mining, the algorithm finds that there are no candidate 3-sequences.

1.4 Frequent-Itemset and Frequent-Sequence Mining

Researchers have created many algorithms for both frequent-itemset and frequent-sequence mining. The majority of these algorithms are *serial algorithms*, that is, they only use one processor on one computer. In an effort to achieve lower runtimes, these algorithms perform multiple scans of the database, allowing them to reduce the number of itemsets or sequences the algorithm counts by skipping those itemsets or sequences that cannot possibly be frequent. Researchers achieve this reduction in the number of itemsets to count using the pruning idea from the Apriori algorithms, which states that if an itemset is infrequent, no superset can possibly be frequent. If no pruning is applied, each set of n items from the input database contains $2^n - 1$ sets that the algorithm needs to count, since every item can either be included in or excluded from each counted set, and the empty set is ignored. The same rule holds for sequences and super-sequences. The Apriori algorithm was much faster than counting every possible set, since it reduced the number of sets to count, and frequent set mining research has followed this principle since.

Some situations exist where performing multiple scans of the input database to avoid having to generate all possible sub-sets or sub-sequences is not an option. One such situation is when a user needs to mine from a data stream. Algorithms cannot rewind and perform multiple passes of streaming data. To perform pruning under these circumstances, where some algorithms perform multiple scans of the original database, others capture the contents of the input database or stream into a structure suitable for mining. These algorithms then perform multiple scans of this structure, rather than the original database.

For example, Leung et al.'s CanTree algorithm [LKLH07] reads the contents of a data stream, captures it to a tree-based structure called a CanTree, and then performs multiple scans of the CanTree to find frequent itemsets. Hualei et al. [HSJ$^+$08] performed the same process with their NC-Tree structure. Kholghi et al. [KHK10] recently performed a survey of stream mining techniques, and found that researchers used a structure to capture the data stream in every algorithm they came across. Similarly, researchers sought to reduce the cost of reading from disk when mining from databases rather than streams. Tanbeer et al. [TAJL09] proposed a frequent-itemset mining algorithm that reads a database once, converts it to a tree-based structure called a CP-tree, and then performs repeated scans of the CP-tree to find frequent itemsets. Zhang et al. [ZLLW09] performed recursive scans of their Index Array structure to find frequent itemsets, and Liu and Yang [LY09] performed recursive scans of their canonical unordered tree structure to achieve the same goal.

Another topic in computer science that is currently important, thanks to technological advances and the ever-increasing number of computers worldwide, is the study of parallel computing. Businesses and research institutions typically have more than one computer available. Since processor manufacturers Intel® and AMD™ each released their first multi-core desktop processors in April 2005, effectively combining multiple processors into one chip, desktop computers having more than one available processor has become the norm. Modern data mining algorithms should consider the availability of multiple computers and processors, and determine if making use of these additional resources can improve their performance. Existing serial data mining algorithms, both those that mine from databases and from data streams, mine the data using one computer. Where the database or data stream supplies data at a higher rate than the computer can mine it, parallel algorithms could distribute the data between multiple computers, increasing the number of input rows handled per second. This requires a basic understanding of parallel computing.

1.5 Parallel and Distributed Computing

Ghosh [Gho06] defined *Distributed memory computing* as multiple computing processes running in disjoint address spaces and communicating with each other to achieve a collective goal. Researchers and software developers commonly use the Message Passing Interface (MPI) [Mes] to handle communication in applications running on distributed memory systems. Since communicating between computers in a distributed memory system is slower than accessing local memory, MPI applications must strike a balance between the amount of communication required and the amount of work done on each computer. For example, if multiple computers were running the Apriori algorithm, no communication would be required if they all ran the entire algorithm independently. In this case, all computers would perform all of the work, and there would be no performance benefit.

Another performance constraint to consider in MPI applications is *barriers*, points in the code that all computers must reach before any of them can continue. Computers that reach a barrier first sit idle while they wait for all of the other computers to reach the barrier as well. This leads to the idea of *load balancing*, which is an attempt by a distributed computing application to divide the work between all of the computers in such a way that all computers reach barriers, including the implied barrier at the end of the programme, at the same time. A perfectly balanced load would leave no computer idle, waiting for another computer, at any point.

Since multiple processors in a single computer are becoming increasingly common, as I mentioned in Section 1.1, *shared memory computing* is an important concept to consider. Chapman et al. [CJv07] define a shared memory computer as a system with multiple processors sharing the same random access memory (RAM) between them. Processors in a shared memory computer do not require messages to pass data from one processor to another, since each processor has access to all of the data. The authors use OpenMP [Opeb], a multi-platform API for developing shared memory applications, to take advantage of these computers.

A challenge arises when multiple processors want to update the same piece of data at the same time. In situations where this is possible, programmers use *locks*, which are mechanisms that ensure only one processor can modify data protected by a lock at a time, and *critical sections*, which are sections of code that only one processor can enter at a time. Similar to barriers in distributed memory computing, waiting for a lock or a critical section can leave a processor idle and become a performance constraint. Another challenge and possible performance constraint results from each processor in a shared memory computer having its own cache and register sets. When a processor makes changes to data from shared memory, it may only change the copy in its cache, rather than in RAM. Using the *flush* instruction, which ensures that each processor writes a specific variable, or all of its cached data to RAM, before reading any shared memory solves this. This can reduce expected performance benefits when compared to a serial algorithm where processors would use cached data frequently before writing back to memory.

1.6 My Research

I have explored using multiple computers with different hardware specifications to perform frequent-sequence mining. Consider the example distributed memory computer in Figure 1.1, consisting of a database, which contains the data that my algorithm will mine, and a number of computers that my algorithm will run on. The computers have a mix of parallel hardware, such as multiple processors, multi-core

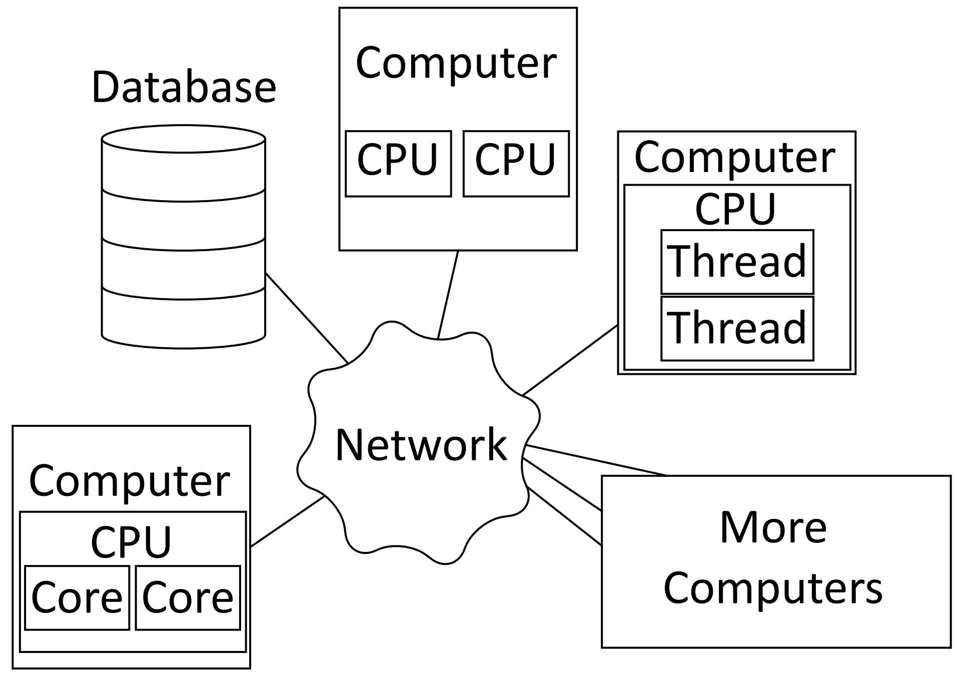


Figure 1.1: Example of a distributed memory system

processors, processors with multiple thread units, and computers combining any of the aforementioned features. Since the data starts in one location, the database, and my algorithm must first retrieve the data onto the computers that it is running on, the database is a potential performance bottleneck. With this in mind, some key questions I will answer are:

1. “Can I mine frequent-sequences on a distributed memory computer using only one scan of the original database?”

2. “Can I avoid transforming the database and scanning said transformed database multiple times?”

3. “Can I limit the amount of communication between computers so that my algorithm scales well?”

4. "Can I make use of multiple processors, cores, and thread units to reduce the runtime of my algorithm?"

My **M.Sc. thesis statement** is as follows:

> I aim to develop an algorithm that mines *frequent-sequences* in one scan of a central database. The algorithm will avoid transforming the original database and scanning the transformation multiple times. It will use distributed and shared memory computers to generate all $2^n - 1$ sequences from each database row to find all frequent sequences. My goal is to complete as much of the mining process as possible in the time spent reading the input data from the central database, which is a required step, and force this to be the performance bottleneck.

To achieve my research goal, in Chapter 2, I review the existing work related to my topic, starting with parallel computing research. I look at methods of using shared- and distributed-memory computers simultaneously, the effectiveness of multiple thread units, and methods of balancing load between computers. I also review and categorize existing frequent-itemset and frequent-sequence mining algorithms that use these technologies, and present their strengths and weaknesses. Finally, I use the combination of existing algorithms and parallel computing research to create the ideal framework for an algorithm meeting my research goals.

Using the criteria for this ideal framework, in Chapter 3, I design my frequent-sequence mining algorithm, which uses multiple shared-memory computers with different specifications. I create two methods of mining each row as my algorithm reads it from the database, without requiring any intermediate structures, and show how these methods make use of multiple processors on shared-memory computers. I also show how my algorithm uses multiple computers, and design two methods of collecting the results. Finally, I present my load-balancing method, which divides the work between computers.

In Chapter 4, I report the results of a series of experiments that I performed on my algorithm. I measure the amount of time and memory my two mining and two collection methods require as factors change. I vary the number of processors and thread units used on a single computer, the number of rows in the input database, the row length of the input database, and the number of computers used. I also measure the effectiveness of my load-balancing algorithm, and compare my result collection algorithms.

Finally, I present my conclusions in Chapter 5. I summarize the goal of my research, and answer each of the key questions I asked in Section 1.6. I then review the opportunities that my research provides for future work.

Chapter 2

Related Work

I am focusing my research on data mining, but it also draws from other areas in computer science. Since I am designing a frequent-sequence mining algorithm using heterogeneous distributed computers, I start by exploring existing parallel and distributed computing research. The specific parallel and distributed computing areas I focus on are the effects of mixing shared- and distributed-memory parallelization, the benefits of current processor technology, and methods of balancing the load in a heterogeneous distributed-memory system.

In this chapter, I also review several distributed frequent-sequence and frequent-itemset mining algorithms, their strengths that I seek to preserve, and their drawbacks that I seek to avoid. Using these strengths and weaknesses, I identify the structure of the ideal data mining algorithm to meet my goals, as well as its requirements and benefits.

2.1 Shared- and Distributed-Memory Computing Technologies

Along with the basics of parallel and distributed computing, as I introduced in Section 1.5, my algorithm makes use of some additional research. Rather than using either shared- or distributed-memory computers, I make use of a combined environment, where each of the distributed computers has multiple processors. I start this chapter by researching models to combine these two parallelization strategies.

As well as multi-core and multi-processor computers, some processors, such as Intel®'s processors with Hyper-Threading technology, provide multiple thread units on each core. I research the effectiveness of this technology to use it in my algorithm in Section 2.1.2.

Finally, I need a method to balance the amount of work each computer performs. To develop this, I first research other author's methods of load balancing, and the effectiveness of these methods, in Section 2.2.

2.1.1 Hybrid MPI and OpenMP

Since I perform data mining on multiple computers, each potentially with multiple processors or threads, I need to explore methods of making use of this hardware. Rabenseifner et al. [RHJ09] researched methods of making use of multiple computers, each with multiple cores, and came up with four possible implementation models. Their first model, which uses purely MPI, creates a discreet memory space for each processor on every machine, and communicates between processors by sending mes-

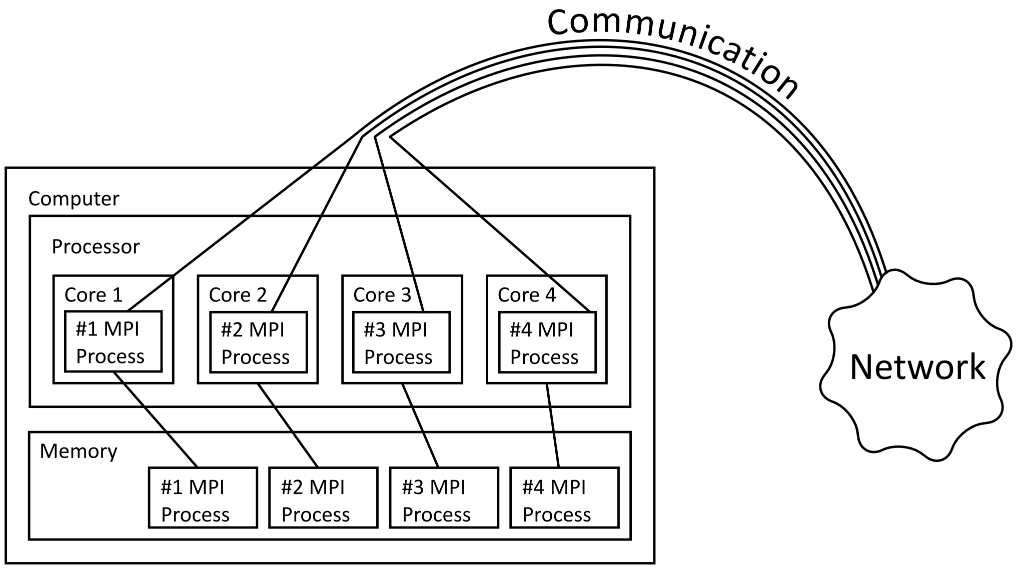


Figure 2.1: Pure MPI model on a single computer

sages. This model works for situations where processors do not require data from others, but is not ideal for a problem such as data mining, where all processors need access to the data mining structures. Since the Pure MPI approach does not share memory between processors, each processor would need to create and maintain its own data structures, increasing both memory usage and the amount of processes that must communicate. Figure 2.1 shows a single computer in the Pure MPI model, with a separate MPI process on each processor core, a separate block of memory in use by each process, and communication between processes and other computers using message passing.

To allow computers to share the memory between their processors, the authors presented two hybrid approaches using MPI and OpenMP, which they called Hybrid Master and Hybrid Overlap. Algorithms based on the Hybrid Master model alternate between two separate stages, one to communicate between nodes with MPI, and one

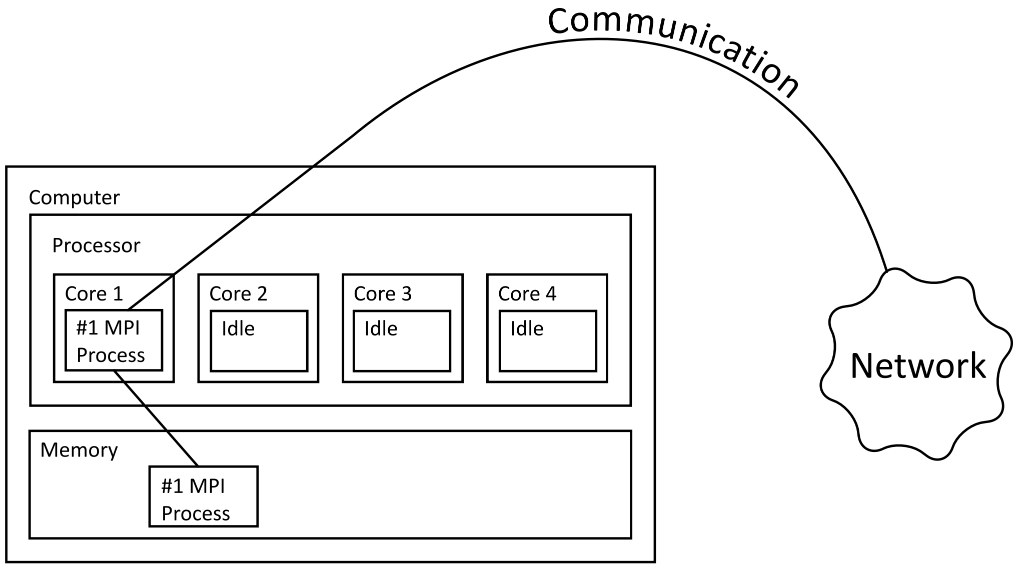


Figure 2.2: The Hybrid Master model during the communication stage

to perform local computation with OpenMP. In the communication stage, algorithms use one processor core to perform communication between computers, solving the communication overhead problem of the Pure MPI model by limiting the number of communication streams per computer to one. I show this in Figure 2.2. Unfortunately, this means that all other processor cores are idle during communication.

When communication is complete and computation is required, algorithms create an OpenMP thread on each processor core, as shown in Figure 2.3. These threads run in parallel and all have access to the same block of memory, resolving the issue of the Pure MPI model where each core has its own block of memory, but introducing a new problem where all OpenMP threads must complete before communication can take place. If one OpenMP thread takes longer than the others do, the Hybrid Master model leaves the other cores idle until the longest thread completes. This approach is ideal when the only communication between computers is during the initializa-

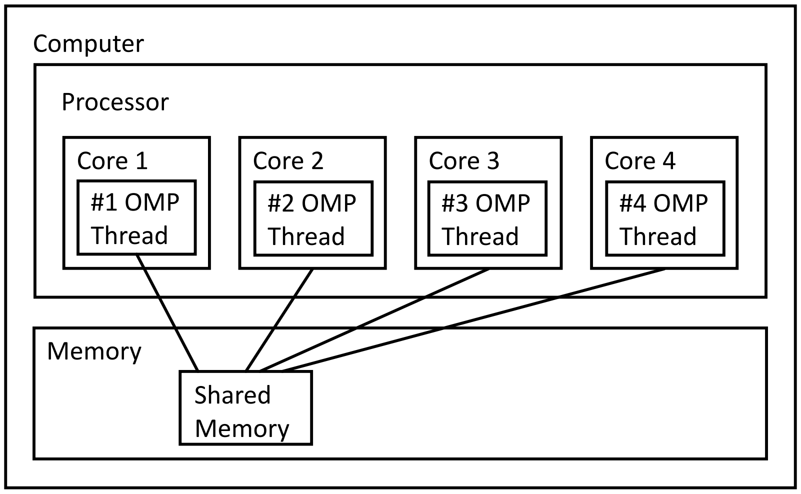


Figure 2.3: The Hybrid Master model during the computation stage

tion and finalization of an algorithm, but not when communication interleaved with the computation is required, since this causes the algorithm to flip between stages frequently.

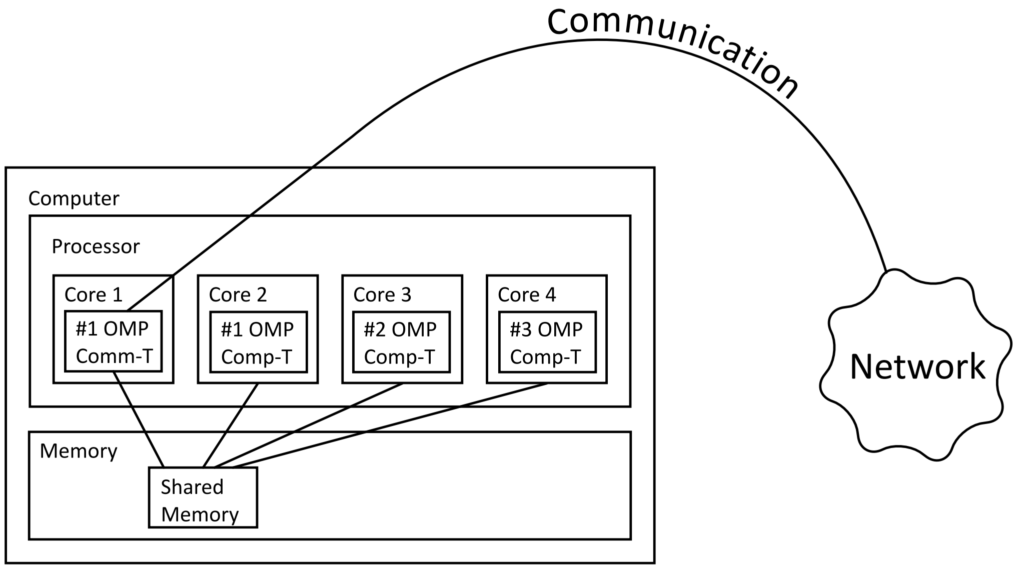


Figure 2.4: The Hybrid Overlap model

The Hybrid Overlap model pushes MPI communication into the OpenMP threads, and rather than using two stages, it uses two thread types – communicating threads and computation threads. If there is little communicating to be done, algorithms using the Hybrid Overlap model have the option to reduce the number of communicating threads to one, which is the minimum so that it is ready to receive communication. This leaves the rest of the cores free to run computation threads. If the amount of communication increases and saturates the available communication threads, Hybrid Overlap algorithms have the option to switch an additional core from a computation thread to a communication thread, increasing the communication capacity. This method eliminates the problems from the Hybrid Master model, where cores are idle during the communication stage or while waiting for other cores to finish the processing stage, and is ideal where the ability to communicate and process simultaneously are required. The drawback to this model is that cores assigned to communication are unavailable to perform computation. Figure 2.4 shows the Hybrid Overlap model with one core running a communication thread, and three cores running computation threads.

The final approach Rabenseifner et al. presented is the Pure OpenMP model, which uses a set of compiler add-ons to allow computers to treat distributed memory like shared memory. This model essentially extends the OpenMP flush operation. Whenever an algorithm accesses data from shared memory, it must not only flush that data from the local processor caches to memory, but also flush that data between all computers. In algorithms that require shared variables, or especially shared data structures that require a flush of all shared memory rather than a specific variable, the amount of communication overhead this model adds quickly becomes much greater than the hybrid approaches. This simplifies programming algorithms for distributed-memory environments, but is very specialized in the areas it is applicable to due to its drawbacks.

Of the four presented models, the *Hybrid models* are the best fit for data mining applications, which require large data structures to either mine from or store results to. Using OpenMP in the hybrid models shares these structures between all local processors to reduce memory usage and communication. Attempting to share these structures between all computers with the Pure OpenMP model would require flushing the entire structure between all computers any time a processor read from it, which would cause far too much communication to be beneficial to overall performance.

2.1.2 Effectiveness of Multiple Thread Units

Along with multiple processors and processors with multiple cores, modern Intel® processors support multiple thread units per core, in the form of their Hyper-Threading technology [Int]. Liao et al. [LLHC08] and Curtis-Maury et al. [CMDAN08] independently tested the effectiveness of this technology, and compared it to multi-core computers, with Liao et al.'s computers running Linux® kernel 2.6.3, and Curtis-Maury et al.'s running Linux® kernel 2.4.25.

In both papers, the researchers found that where increasing the number of processor cores increased performance in all of their parallel benchmarks, increasing the number of thread units had varied results. Some tests had an increase in performance as expected, mainly those where the data required could be stored in the processor's cache or memory access was limited, but memory-intensive tests lead to performance degradation due to competition for memory access. Both sets of researchers also found that, when using a multi-core processor with Hyper-Threading on each core and setting the number of threads to match the number of cores, the Linux® kernel was likely to execute the threads on as few cores as possible. This consistently led to poorer performance than forcing the threads to spread over each core. They both conclude that the Hyper-Threading aware schedulers in their respective Linux® kernels are suboptimal.

From these results, it seems that if my goal of forcing the performance bottleneck of a data mining algorithm to be reading from the database, rather than the memory bottleneck typically seen in existing algorithms, is successful, Hyper-Threading will be quite beneficial. I will perform my experimentation using Hyper-Threading on a Microsoft® Windows® based cluster, rather than Linux®, so it will be interesting to see if Hyper-Threading is beneficial to an algorithm with high memory access rates on this operating system.

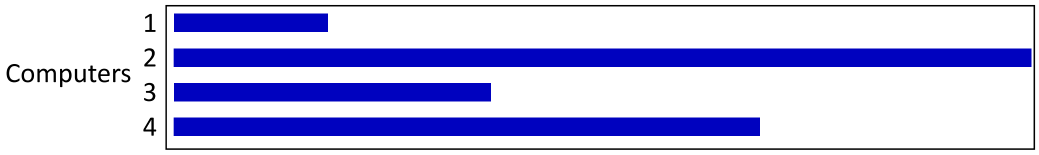
2.2 Load Balancing

Since the goal of my algorithm is to run on all available computers, it needs to be able to run on a distributed system made up of a range of hardware specifications. This means that assigning work to each computer is not a simple matter of splitting it evenly over a number of identical computers, but requires a method to balance the amount of work assigned to each computer. The goal of a balanced load is to have each computer complete its processing at the same time.

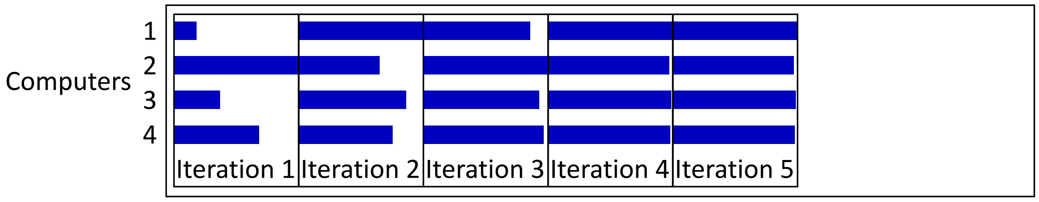
Utrera et al. [UCL08] researched balancing load when there are more processes to run than the number of available processors in a distributed memory system where all computers have identical specifications. This research is not directly applicable to my target system, but the authors raise an important point. A load balancing mechanism not only needs to consider the ideal division of work, but also must factor in the amount of communication and synchronization required to perform load balancing itself. The load balancing algorithm the authors present samples each process to predict its total required runtime, and uses this required runtime to balance the load. In cases where the processes to run only have mild imbalance when run in the order they arrive in, the overhead of their predictive algorithm results in it having longer runtimes, even though it has a highly balanced load.

Galindo et al. [GABC08] presented a similar idea to Utrera et al. to balance load on a distributed system, but one made up of computers with different specifications, and where the amount of data to be processed determines the balance of load. Their algorithm starts with the data divided uniformly between computers and measures the amount of time to run a set number of iterations on each computer. The algorithm then uses these runtimes to compare the computers to each other, and modifies the portion of data and the number of iterations to measure for each computer accordingly. The algorithm repeats itself, changing the portions after each set of iterations unless the difference between the minimum and maximum runtimes is below a balance threshold. Their results show that this method gives a balanced load.

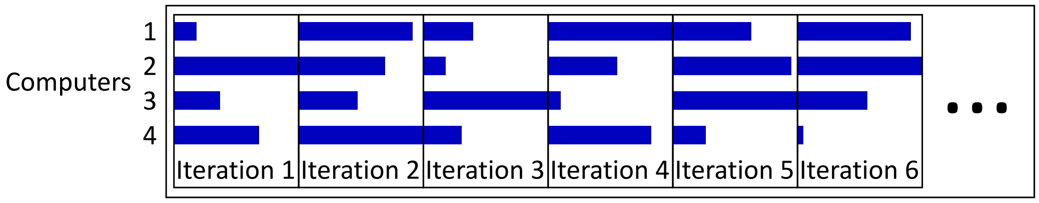
The drawback is that it requires a barrier after each set of iterations to collect time measurements. In tests where the data to be processed is fairly uniform, the load



(a) Runtimes with equal portion sizes



(b) Runtimes with dynamic balancing and uniform data



(c) Runtimes with dynamic balancing and non-uniform data

Figure 2.5: Theoretical runtime graphs

is balanced after a few iterations, meaning each computer reaches the barrier after each set of iterations at the same time, and there are gains of over 50% compared to assigning each computer an equal portion of the data. In the case where the data to be processed is not of uniform computational requirements, there is a rebalancing after every set of iterations, meaning that there is idle time waiting for all computers to complete after each set that is introduced by the load balancing algorithm itself. While the runtime of each computer still ends up more balanced than partitioning the data equally amongst processors in this case, the total runtime was over 15% worse than the longest unbalanced time, because of the overhead added by the balancing algorithm.

Consider the theoretical runtime graphs in Figure 2.5, representing the runtimes of a distributed memory algorithm running on four computers. The first graph represents the runtime with the data partitioned equally amongst the computers. If the data is uniform, it shows that the first computer is the fastest, if the data is not uniform, it may be that the first computer is the fastest, or that it happened to receive data requiring less computation than the other computers received.

The second graph shows the expected runtimes if the data distribution was uniform and the first computer is the fastest. On the second iteration, the load balancing algorithm gives the most data to the first computer, since it had the lowest previous runtime, the second most to the third computer, and so on. It proceeds in the manner until iteration four, where the loads are balanced. The time the load balancing algorithm spends waiting for other computers after iterations is inconsequential once the load is balanced. The balanced load greatly reduces the overall runtime compared to the equal portion size runtime since the faster computers are not sitting idle for a large portion of the time.

The third graph shows what could happen if the data distribution was so non-uniform in terms of complexity that the resulting runtimes from one iteration are not a predictor of the runtimes for the next. All iterations are unbalanced, the balance does not improve over time, and the overall runtime increases compared to the equal portion size runtimes since the balancing algorithm leaves computers idle after all iterations. The only reason the third graph will end with all computers completing execution closer to the same time than the equal portion size runtimes is that all computers wait for one iteration to end before beginning the next. The only differences in the final runtimes come from the final iteration.

I expect this method of load balancing, using the runtime of the algorithm to measure each computers performance, would have the same results when applied to data mining. It would create a well-balanced load at low overhead cost in all but the extreme cases where one row in the input dataset has no correlation with the next.

2.2.1 Shared- and Distributed-Memory Computing Technologies Summary

I found that the Hybrid models of combining MPI with OpenMP can take full advantage of shared- and distributed-memory computers, gaining the benefits of both libraries. Shared-memory applications generally benefit from the use of Hyper-Threading, but less so in comparison to using more cores or processors.

For load balancing, measuring the runtime of an algorithm for a portion of a dataset is a good predictor of the runtime over the entire dataset. Load balancing algorithms have to consider the amount of overhead they require themselves. They must strike a balance between the amount of communication and processing they require and the level of balance they achieve. Algorithms that require high overhead to produce a perfectly balanced load end up taking more total runtime than algorithms requiring little overhead to produce a near-balanced load.

Table 2.1: Example sequential database

Id	Sequence
1	$\langle$ a, c, f, g, e $\rangle$
2	$\langle$ a, f, c $\rangle$
3	$\langle$ b, e, d, a $\rangle$
4	$\langle$ f, a, b, e $\rangle$
5	$\langle$ d, c, a, f, g $\rangle$
6	$\langle$ d, a, f $\rangle$

2.3 Distributed Frequent-Pattern Mining Algorithms

Researchers have used variations of three different techniques to convert serial frequent-sequence and frequent-itemset mining algorithms into distributed and parallel versions. In general, the first technique divides the input dataset into non-overlapping partitions amongst each computer, and then performs a modified serial frequent-pattern mining algorithm, exchanging counts after each level. I categorized these as *synchronous partitioned data algorithms*. The second technique performs the same data distribution, but only collects counts at the end of the sequential algorithms, and must then re-scan the input database to find global counts. I have categorized these as *asynchronous partitioned data algorithms*. The third technique copies the entire input dataset to each computer, and then divides the work by telling computers to use a serial algorithm to mine *only* for sequences starting with a specific item, collecting the results when the serial algorithm completes. I categorize these as *asynchronous duplicated data algorithms*.

Table 2.2: Database with projection offsets

Id	Sequence	a	b	c	d	e	f	g
1	⟨ a, c, f, g, e ⟩	2	∅	3	∅	$	4	5
2	⟨ a, f, c ⟩	2	∅	$	∅	∅	3	∅
3	⟨ b, e, d, a ⟩	$	2	∅	4	3	∅	∅
4	⟨ f, a, b, e ⟩	3	4	∅	∅	$	2	∅
5	⟨ d, c, a, f, g ⟩	4	∅	3	2	∅	5	$
6	⟨ d, a, f ⟩	3	∅	∅	2	∅	$	∅
Counts:		6	2	3	3	3	5	2

2.3.1 Synchronous Partitioned Data Algorithms

She et al. [STL+05] presented two distributed memory versions of the PrefixSpan [PHM+04] serial frequent-pattern mining algorithm, both using the synchronous partitioned data algorithm (SPDA) technique. The serial PrefixSpan algorithm finds frequent sequences by recursively forming databases containing only sequences starting with a specific prefix in a breadth first manner. These are projected databases. Consider mining all sequences with a minimum support of 3 from the database in Table 2.1. The first step PrefixSpan performs is to count the singleton items. While it is counting singletons, it also tracks *projection offsets*, that is, the starting point in the sequence if it was included in a projected database for a specific singleton. The authors call this technique *pseudo-projection*, and use it to avoid having to copy a portion of each sequence to a new database when projecting. An offset of 1 represents the first item in the sequence, an offset of 2 represents the second item, and so on. I have shown the database with the corresponding projection offsets and counts in Table 2.2, with the symbol $ indicating a prefix that exists but has no items following it, the symbol ∅ indicating that the prefix does not exist in the database sequence, and the last row representing the singleton counts.

Table 2.3: *a*-projected database

Id	Sequence	singletons	ac	ad	ae	af
1	⟨ a, c, f, g, e ⟩	...	3	∅	$	4
2	⟨ a, f, c ⟩	...	$	∅	∅	3
3	⟨ b, e, d, a ⟩	...	∅	∅	∅	∅
4	⟨ f, a, b, e ⟩	...	∅	∅	$	∅
5	⟨ d, c, a, f, g ⟩	...	∅	∅	∅	5
6	⟨ d, a, f ⟩	...	∅	∅	∅	$
Counts:		...	2	0	2	4

PrefixSpan continues in a breadth first recursive manner, moving from left to right across the offset columns, and ignores columns with counts below the minimum support value. It adds new columns representing new projected databases. Table 2.3 shows the *a*-pseudo-projected database. Each sequence in the database effectively starts at the position corresponding to the projection offset from column *a* in Table 2.2. There are no *ab* or *ag* position offsets calculated, since the singletons *b* and *g* had counts less than the minimum support (3). PrefixSpan determines which projections it needs to track offsets for using the same method as the Apriori algorithms, that is, PrefixSpan only finds the offsets for a k-item sequence if all $(k-1)$-item sub-sequences are frequent.

She et al. designed their SPDA versions of PrefixSpan for homogeneous distributed-memory systems, that is, systems in which all computers have the same specifications. Their first version starts by reading a non-overlapping partition of the database on each computer, creating the singleton pseudo-projection columns as it reads the database, as PrefixSpan does. Once it completes the singleton counts on all computers, it collects the total counts from all computers onto one computer, and then sends the totals back to the rest of the computers to use for the next level.

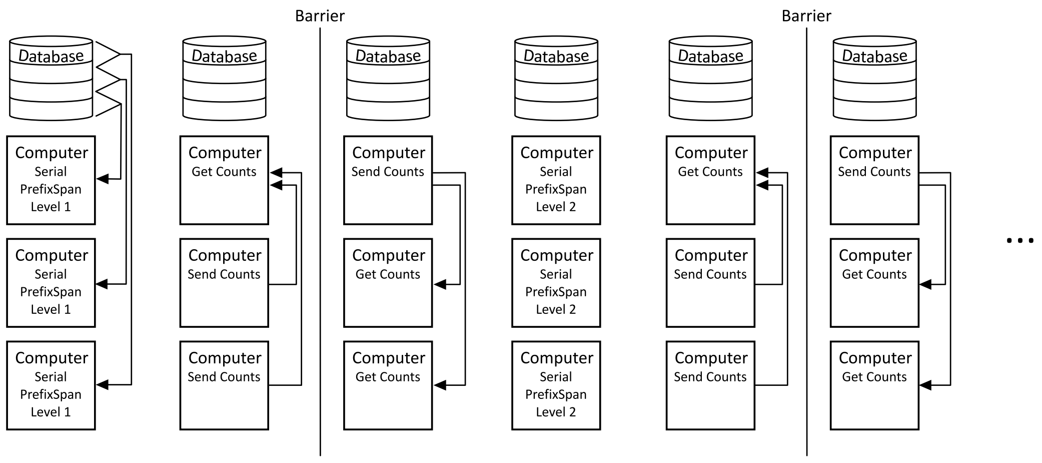


Figure 2.6: Stages of an SPDA; She et al.'s first distributed PrefixSpan

Figure 2.6 shows the stages of She et al.'s first parallel version of PrefixSpan, which are also the stages of the basic SPDA framework. Moving from left to right, the algorithm loads a partition of the database onto each computer, which runs the serial version of PrefixSpan. As each computer completes the singleton counting and pseudo-projecting stage, it sends its counts to the collecting computer. The collecting computer receives local counts and computes global counts. There is a barrier after the count collecting, since the counting computer must receive all other local counts before it can send the global counts, which the algorithm performs in the third stage. Once a computer receives the global counts, it can start performing the second level of the PrefixSpan algorithm, without having to wait for the other computers to complete receiving the global counts. The algorithm then repeats until it completes mining.

The first problem, and most important relative to my research, is that She et al.'s algorithm leaves the database idle after the first stage of the algorithm. There is no possibility of completing the mining process in the same time it takes all of the mining computers to read the database. Another problem is that there are frequent barriers, which, as I mentioned in Section 1.5, hinder scalability. All computers are idle at each barrier point until the last computer completes. The amount of communication required also limits the scalability of the algorithm. As the number of computers increases, the amount of data that it must transfer in each stage increases accordingly. Fang et al. [FZZ$^+$05], and Renjit and Shunmuganathan [RS10], have both applied the same technique used in She et al.'s first algorithm to create distributed-memory versions of the Apriori algorithm for frequent itemset mining. Their algorithms have the same strengths and weaknesses as She et al.'s first algorithm.

Seeing the scalability problems introduced by the number of barriers and the amount of communication, She et al. presented a second version of their distributed PrefixSpan algorithm. Their second version starts the same as the first version, reading a portion of the database on each computer, and performing the first level of the PrefixSpan algorithm. It continues as normal until it reaches a level where it has created at least as many projected databases to mine as there are computers. At this point, the algorithm assigns each projected database that it has not yet mined to a specific computer. Every computer must go through all of its locally projected databases and send them to the computer that will mine them.

Simply mapping the first $> n$ projected databases at the end of a level over n computers would lead to load imbalance. Some projected databases will take much longer to mine than others. To keep the amount of work each computer has to perform balanced, She et al. continue to map projected databases that their algorithm still needs to mine after each level. The balancing mechanism looks at the number of projected databases each computer generated after each level, and re-assigns these

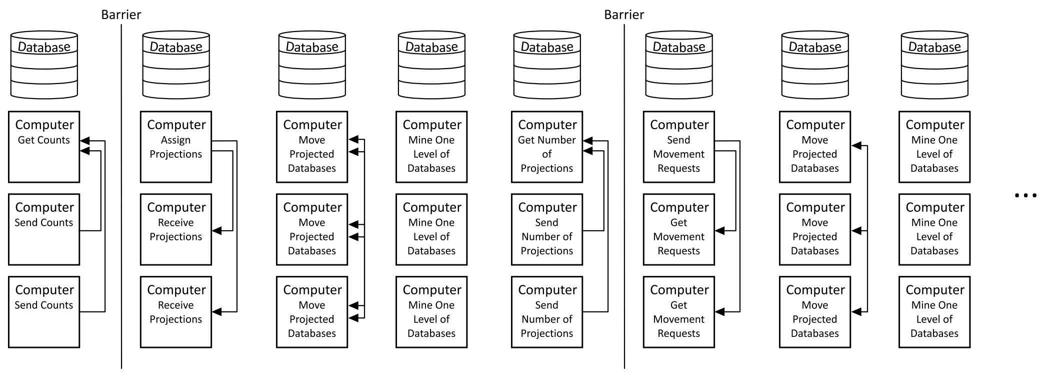


Figure 2.7: She et al.'s second distributed PrefixSpan

databases evenly over each computer, moving as few databases as possible.

Figure 2.7 shows the stages She et al.'s second SPDA PrefixSpan, starting from the point in their first algorithm where it collects the global counts and more projected databases to mine than computers to mine on. At this point, the algorithm on one computer, the *controlling computer*, assigns each of the pending projections to one computer. After assigning the projections, each computer must send the projected databases themselves to their assigned computers. This creates a significant amount of communication since each computer will receive a projected database from every other computer for each projected database assignment.

Once a computer has finished sending and receiving all of its projected databases, it mines one level of each assigned projected database, tracking the number of new projections it creates for the next level. As computers finish a level of mining, they each send the number of next-level projections to the controlling computer. Once the controlling computer has received all next-level projection counts, it balances the load by ensuring each computer has the same number of projected databases to mine. It sends movement requests to all other computers, which may ask the computer to perform no action, receive a number of projections, or send a number of projections to certain computers. Once a computer has received all of its movement requests, it performs the requested tasks, moving projected databases where necessary. Following the database movement, each computer mines the next level, repeating the load balancing steps and mining until there are no more next-level projected databases to mine.

This second algorithm outperforms the first when it does not have to move many databases after the initial assignment. When the load is balanced, the amount of communication is significantly less than the first algorithm requires gathering and broadcasting all counts, since it only has to gather the number of projections at each level. Unfortunately, the load balancing mechanism has the potential to add significant communication overhead, and requires a barrier after each level as in the first algorithm. As Utrera et al. pointed out, researchers must factor overhead from load balancing into the effectiveness of an algorithm. When the load is not balanced, She et al.'s second algorithm must send entire projected databases, including the database row, its counts, and its projection indices, after each level. Paul and Saravanan [SS08] used the same technique as She et al.'s second algorithm to create a distributed-memory version of the frequent-itemset mining Apriori algorithm, using a hash function to map itemsets quickly, but with no load balancing.

All of these SPDAs leave the database idle after reading the data the first time, leaving no possibility for the algorithms to complete the mining process as they read the data. The communication and barriers they require limits the scalability, in regards to the number of computers used, of all of the SPDAs.

2.3.2 Asynchronous Partitioned Data Algorithms

Asynchronous partitioned data algorithms (APDA) aim to scale more efficiently, in regards to the number of computers they use, compared to SPDAs. The focus of these algorithms is the removal of barriers and communication during the mining process. The technique used to create Asynchronous Partitioned Data Algorithms is simple, and it works with any existing serial mining algorithm, using the idea of *local minimum support*. Tanbeer et al. [TAJ09] used the APDA approach to create a distributed mining algorithm.

The key idea in Lakshmanan et al.'s Segment Support Map [LLN00] , that "for an itemset to be globally frequent, it must be locally frequent on at least one segment", is the basis for the idea of local minimum support. This idea applies to distributed memory computers as "for an itemset to be globally frequent, it must be locally frequent on at least one *computer*." Although the Segment Support Map mines frequent itemsets, the concept of local minimum support works, replacing segments with computers, for frequent sequences as well. Equation (2.1) shows how APDAs calculate the portion of rows assigned to computer i:

$$\frac{assignedRows_i}{totalRows} = portion_i \tag{2.1}$$

Equation (2.2) shows how they use this portion to create the local minimum support of computer i:

$$minSup * portion_i = localMinSup_i \tag{2.2}$$

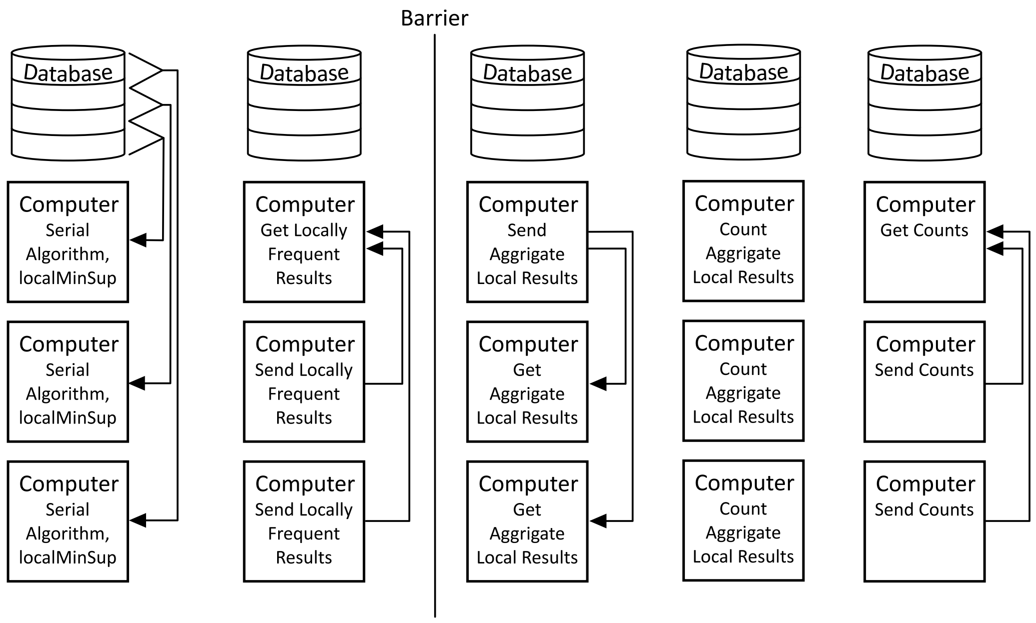

Barrier
Database
Database
Database
Database
Database
Computer Serial Algorithm, localMinSup
Computer Serial Algorithm, localMinSup
Computer Serial Algorithm, localMinSup
Computer Get Locally Frequent Results
Computer Send Locally Frequent Results
Computer Send Locally Frequent Results
Computer Send Aggregate Local Results
Computer Get Aggregate Local Results
Computer Get Aggregate Local Results
Computer Count Aggregate Local Results
Computer Count Aggregate Local Results
Computer Count Aggregate Local Results
Computer Get Counts
Computer Send Counts
Computer Send Counts

Figure 2.8: Stages of an APDA

Since the partitions assigned to each computer are non-overlapping, it follows that the sum of all assigned rows equals the number of total rows, the sum of all portion sizes equals one, and the sum of all local minimum supports equals the global minimum support. Consider the case where each computer has a local minimum support, $localMinSup_i$, and a sequence occurs $localMinSup_i - 1$ times on each computer, which is the most it can occur without being locally frequent. Since the sum of all local minimum supports equals the global minimum support, but the sequence occurs one less often than the local minimum support on each computer, the sum over all computers is $minSup - numComp$. The global count cannot be greater than or equal to the minimum support, since the number of computers must be greater than zero, so no sequence that isn't locally frequent on at least one computer can be globally frequent.

As with the SPDAs, and shown in Figure 2.8, APDAs start by reading a non-

Table 2.4: A database partitioned evenly between two computers

Computer 1	Computer 2
$\langle$ a b c $\rangle$	$\langle$ a b c $\rangle$
$\langle$ a b c $\rangle$	$\langle$ d e f $\rangle$
$\langle$ a b c $\rangle$	$\langle$ g h i $\rangle$

overlapping partition of the database on each computer. This is as far as the similarities go. Each computer in an APDA creates a local minimum support based on the amount of data it is assigned. They then independently run a serial frequent itemset or frequent sequence mining algorithm until they discover all locally frequent itemsets or sequences. As each computer finishes running its serial mining algorithm, it sends the results back to the controlling computer. The controlling computer collects locally frequent sequences from all computers and builds a list of potentially globally frequent sequences. From Lakshmanan et al.'s work on the Segment Support Map [], any locally frequent set is potentially globally frequent.

Once the controlling computer receives all locally frequent sequences, it must send the set of potentially globally frequent sequences to each other computer. The computers must now re-scan their input databases and count all of the potentially globally frequent sequences. The algorithm then collects the results of the second database scan, and outputs the globally frequent sequences.

Re-scanning the database is necessary because the serial algorithms prune any locally infrequent sequences and do not mine supersets of them. Consider the database partitioned between two computers in Table 2.4 with a minimum support of 4. Each computer has 3 of 6 total rows, so the portion sizes are $3/6 = 0.5$ and the local minimum supports are $4 * 0.5 = 2$. The first computer mines the frequent sequences

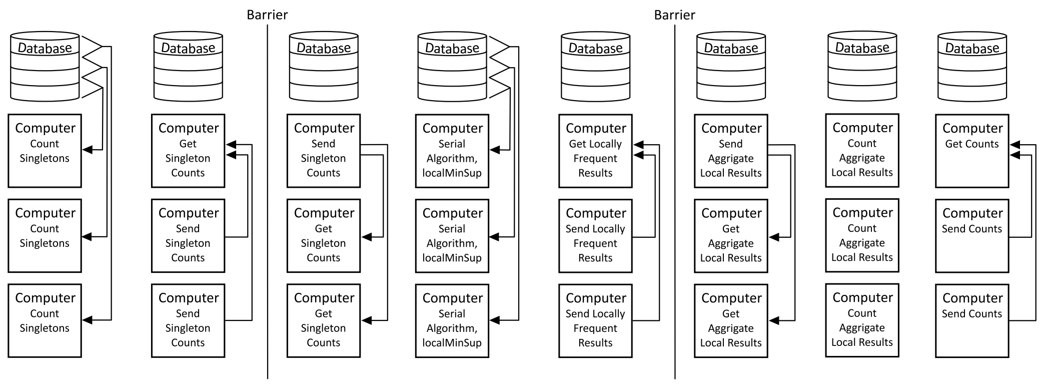

Barrier
Barrier
Database
Computer
Count Singletons
Get Singleton Counts
Send Singleton Counts
Serial Algorithm, localMinSup
Get Locally Frequent Results
Send Locally Frequent Results
Send Aggrigate Local Results
Get Aggrigate Local Results
Count Aggrigate Local Results
Get Counts
Send Counts

Figure 2.9: Tanbeer et al.'s modified APDA

$\langle\ a\ \rangle$, $\langle\ b\ \rangle$, $\langle\ c\ \rangle$, $\langle\ a\ b\ \rangle$, $\langle\ a\ c\ \rangle$, $\langle\ b\ c\ \rangle$, $\langle\ a\ b\ c\ \rangle$, all with a count of three. The second computer finds that all singletons have a count of one and stops mining with no locally frequent sequences. If an APDA collects locally frequent sequences, but neglects to recount the globally frequent sequences, it would return the result from the first computer as an incorrect global count. It cannot simply request all of the counts that were locally frequent on the first computer from the second, since the second computer only counted the singletons and stopped. A re-scan of the input database is required to achieve the correct result.

APDAs require far fewer barriers than SPDAs during the mining process, only one barrier is required after the locally frequent results are collected compared to a barrier after every level, and computers do not need to communicate during the mining stage. Tanbeer et al. [TAJL09] found that their frequent-itemset mining algorithm using the APDA approach scaled much better when the number of computers increased compared to SPDA algorithms. The authors designed their algorithm to run on identical computers, so they do not worry about load balancing, but partition the database evenly amongst the computers. They also presented one modification to the

APDA structure, collecting the global singleton item counts on each computer before running the serial algorithm, to prune itemsets that are locally frequent but cannot be globally frequent because they contain an infrequent singleton. This adds another barrier and increases the communication of their APDA algorithm, but increases the accuracy of the singleton item pruning, potentially reducing the size of the locally frequent results that the algorithm aggregates and recounts. Figure 2.9 shows the structure of Tanbeer et al.'s modified APDA.

APDAs ability to mine as they receive input data is dependent on the serial algorithm they use. Since existing serial algorithms perform repeated scans of the database, or repeated scans of a structure capturing the contents of the database, they limit the abilities of the APDA technique. This technique leaves the central database idle through the serial mining process, and the subsequent steps, meaning it cannot complete mining in the time it takes to read the database. The main problem with the APDA approach is that locally frequent itemsets or sequences are not necessarily globally frequent, and locally infrequent itemsets or sequences may not be globally infrequent. Having each computer re-scan its partition of the input database and count every potentially frequent itemset or sequence is a time consuming task.

2.3.3 Asynchronous Duplicated Data Algorithms

Asynchronous duplicated data algorithms (ADDAs), such as Cong et al.'s distributed version of PrefixSpan [CHHP05] , Cong et al.'s distributed version of BIDE [CHP05], and El-Hajj and Zaiane's distributed version of the frequent-itemset mining FP-

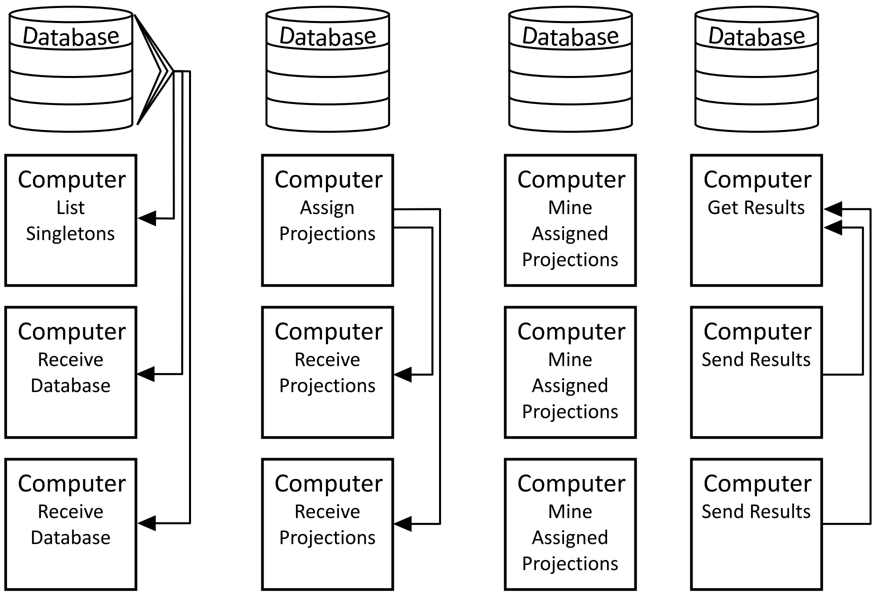

Database
Database
Database
Database
Computer List Singletons
Computer Receive Database
Computer Receive Database
Computer Assign Projections
Computer Receive Projections
Computer Receive Projections
Computer Mine Assigned Projections
Computer Mine Assigned Projections
Computer Mine Assigned Projections
Computer Get Results
Computer Send Results
Computer Send Results

Figure 2.10: General ADDA structure

tree [EZ06] , follow a methodology that is much different from either of the partitioned data techniques. The goal is to provide a distributed framework for existing serial frequent-itemset or frequent-sequence mining algorithms to run in where they do not need to exchange counts during the mining process, as in SPDAs, and do not need to re-scan the input databases, as in APDAs. The key requirement for ADDAs is that every computer reads, and stores or captures, the entire input database. They distribute the work not by partitioning the input data, but rather, they assign different prefixes to each computer to mine. This is similar to She et al.'s second distributed algorithm.

Figure 2.10 shows the structure of a general ADDA. First, the algorithm receives the database on each computer, either storing it as a copy of the input database, or capturing it in a data structure. The controlling computer creates a list of singletons, and their counts, that it will assign to other computers. In the most basic ADDAs, the controlling computer can either assign all projections at this point, or allow computers to request another projection as they complete their current one. Each computer mines all of its assigned projections, and when there are no more projections for it to mine, it sends its results back to the controlling computer.

ADDAs following this technique do not require any barriers during the mining process. Each computer can proceed to the next step as it receives its required information. The amount of communication is low, since the serial mining algorithm used to mine the assigned projections requires none. Unfortunately, there are two large performance constraints in this type of algorithm. The first performance constraint is that the input cost of reading the entire dataset on each computer is high. El-Hajj and Zaiane found that ADDAs do not scale well in regards to the number of computers when the input dataset is large and stored on a shared disk. They reduce the input cost by reading a partition of the database on each computer, transforming it to a tree-based structure, and exchange the trees between all computers to obtain the total database. Cong et al. assumed that each computer starts with the entire input database stored locally in both of their algorithms to avoid this cost.

The second performance constraint is that the amount of mining each singleton requires may be vastly different, leading to load imbalance. El-Hajj and Zaiane balanced load by sorting the singletons by the number of two item projections they have into non-increasing order in regards to size, and assigning two databases, the largest and the smallest, to computers as they require more work. This load balancing scheme can be used on heterogeneous computers, but does not factor in performance differences between computers when performing projection assignments. It also only

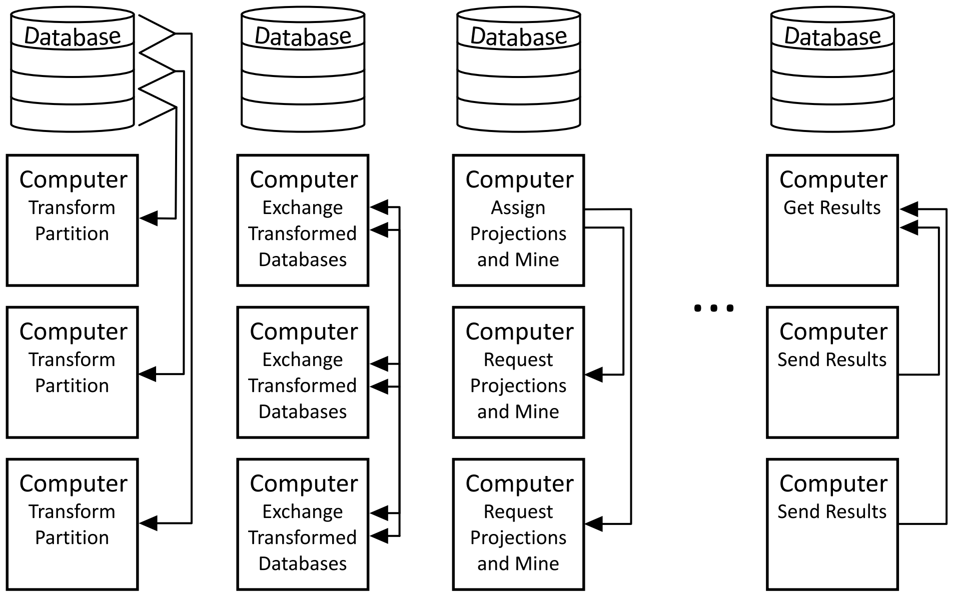


Figure 2.11: El-Hajj and Zaiane's FP-tree based ADDA

divides the singleton items to balance the load. In cases where a few singletons require a large percentage of the work, especially if the algorithm assigns these singletons to slower computers, the load will be unbalanced. The number of singletons is also a limiting factor in scalability. Since the algorithm initially assigns two singletons to each computer, if there are not at least twice as many singletons as there are computers, the algorithm will leave some computers idle.

Figure 2.11 shows the stages of this algorithm. It starts by capturing a partition of the database on each computer. Once each computer receives and transforms its assigned data, it sends the transformed data to each other computer. When the controlling computer has received all of the transformed databases, it assigns the largest and smallest projected databases to the first computer, the second largest and smallest to the second, and so on. As each other computer completes receiving the transformed databases, they wait to have projections assigned to them, and then start mining using a serial mining algorithm. Any time a computer completes mining its assigned projections, it requests the next largest and smallest projections from the controlling computer. El-Hajj and Zaiane's approach leaves the input database idle after the first stage, meaning it cannot achieve my goal of completing the mining process in near the time it takes to read the input database.

Cong et al. presented two ADDAs, one using PrefixSpan, and another following the same framework but using BIDE as the sequential frequent-sequence mining algorithm. They designed their algorithms to run on homogeneous computers. As I mentioned in the problems with the general ADDA structure, Cong et al.'s algorithms assume that all computers start with the entire input database stored locally, rather than having to read it from a central location. I will present their algorithms to adhere to my requirements, mainly, that the database is stored at a central location.

The focus of Cong et al.'s algorithms is to balance the load in such a way that the number of singletons does not limit scalability, as it does in El-Hajj and Zaiane's algorithm. They achieved this by letting the projected database assignments be not just singletons, but also multiple item prefixes as well. If they kept a global queue of prefixes yet to be processed, assigning them on an as-needed basis, it would work the same as She et al.'s second optimization, which requires a lot of communication to move projected databases along with their assignments.

Cong et al.'s solution is to estimate the amount of time required to mine each singleton by mining a sample dataset. They construct a sample dataset by discarding the last l items from each transaction, where $l = 0.75 * averageRowLength$. They then run a serial frequent-pattern mining algorithm on this sample dataset, using the same minimum support value the user sets for the full dataset, and measure the amount of time taken mining each projection. Any singleton requiring more than 25% of the ideal time for each computer, that is, 25% of the total time to mine all singletons divided by the number of computers, is sub-divided. The sub-division of a singleton is all possibly frequent sequences of length 2 starting with that singleton. The algorithm cannot just assign those sequences that are frequent in the sample dataset, or even just those that exist in the sample dataset, since more may exist and be frequent globally. The only sub-divisions that the algorithm does not need to assign are those that cannot possibly be globally frequent based on singleton counts. The algorithm performs recursive sub-divisions until no database projection takes more than 25% of the ideal time for each computer. Once the algorithm reaches this stage, it assigns all of the globally frequent singletons, or their sub-divisions, amongst the computers so that each computer receives an equal estimated workload. It assigns any singleton or sub-division that is not frequent in the sample a mining time of zero.

Figure 2.12 shows the stages of Cong et al.'s algorithms, modified to work with a central database. In the first stage, each computer retrieves the entire database and stores it locally, and performs singleton counts on an equally sized non-overlapping partition. The computers then send all of the singleton counts to the controlling computer. Once the controlling computer has received all singleton counts, it mines a sample of the dataset using only the controlling computer. The controlling computer then analyzes the time to mine each projection of the sample dataset, and uses these times to assign projections between all of the computers. Once each computer receives all of its assigned projections, it mines from those projections using a serial mining

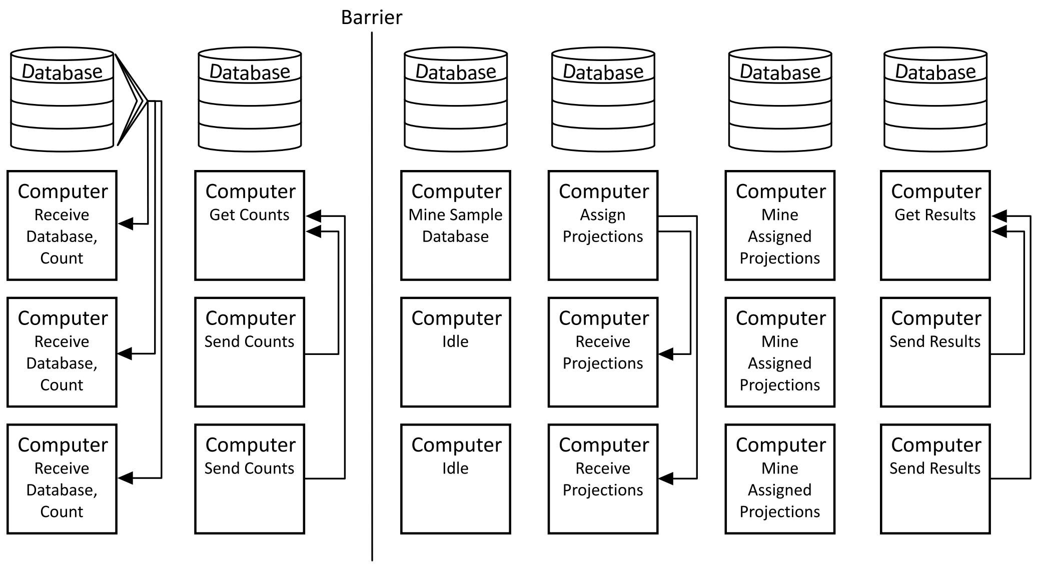


Figure 2.12: Cong et al.'s ADDA framework

algorithm, PrefixSpan or BIDE in this case. Once the serial algorithms are complete, the controlling computer collects the results and the algorithm terminates.

Cong et al.'s algorithm succeeds in removing the singletons as a limitation to load balance and scalability, in comparison to El-Hajj and Zaiane's approach. They break the assigned projections into smaller sub-divisions. Each sub division requires approximately the same estimated mining time. There are some problems with this approach though. First, and foremost relative to the goal of my research, is that the algorithm leaves the central database idle throughout the mining process. Cong et al. also did not take any measures to reduce the input cost of reading the entire database with multiple computers.

Another problem is the method of load estimation. As databases become larger, the size of the sample database increases, as well as the time the algorithm requires mining it. The algorithm only uses the controlling computer to mine the sample dataset, leaving not only the database, but also all other computers, idle during this time. It has to complete the mining of the sample database before it knows the times for the singletons, as their times include mining their sub-projections, so it cannot start assigning work while it estimates.

A related problem, which becomes more pronounced as the database sizes increase, is that the estimation algorithm must assign all potentially frequent projections of length two or more when it performs sub-divisions. An approach such as She et al.'s second SPDA only has to project from globally frequent sequences. No projections that are infrequent in the sample dataset have estimated times. In a database with many frequent sequences, especially if these sequences commonly occur near the tail of database rows, the algorithm will result in a larger number of sequences with no estimated times. Cong et al. found a trade-off between estimation accuracy and the size of the sample dataset. An accurate estimation of a dataset with many frequent sequences occurring at different points within each row requires a larger sample dataset. This compounds the problem of leaving all other computers idle while the controlling computer mines the sample dataset.

2.3.4 Summary of Mining Approaches

Researchers have used three techniques to convert serial frequent-sequence and frequent-itemset mining algorithms into distributed memory versions. The approaches are *synchronous partitioned data algorithms* (SPDAs), *asynchronous partitioned data algorithms* (APDAs), and *asynchronous duplicated data algorithms* (ADDAs). Each type of algorithm has both strengths, and weaknesses, in comparison to the others.

Synchronous partitioned data algorithms are those algorithms that read a partition of the input database on each computer and exchange itemset or sequence counts throughout the mining process. Before any computer begins projections of length k, the SPDA framework ensures that it has the global counts for all itemsets or sequences of length $k-1$. SPDAs read a non-overlapping portion of the dataset on each computer, as opposed to having each computer read the entire dataset, which El-Hajj and Zaiane have shown to have a high input cost that limits scalability. By ensuring that each computer has the global counts, SPDAs never spend time mining an itemset or sequence that is locally frequent but not globally frequent.

The drawbacks of SPDAs are the amount of communication, and the number of barriers, that they require. Implementations such as She et al.'s first algorithm exchange counts after each level of mining. This requires a large amount of communication, since each computer has to send the count of every sequence or itemset of the current level to every other computer. As the number of computers increases, so does the amount of communication, which limits scalability. This is because, as Ghosh stated and I have presented in Section 1.5, communication between computers has a higher cost than accessing local memory.

Because exchanging counts takes place after every level, all computers must wait for the slowest computer to complete one level before moving to the next, which is the performance constraint of barriers as described by Ghosh. She et al.'s second algorithm limits the number of levels that these waits occur after, and reduces the amount of communication synchronizing counts after each level. It does this at the cost of moving projected datasets between computers, which again increases the communication required, and limits scalability. After the initial collection of projected databases onto specific computers, their algorithm moves databases as required to balance load, again with high communication cost. As Utrera et al. have shown, even if a load balancing technique achieves a balanced load, the cost of the load balancing itself can increase the overall runtime in comparison to the same algorithm and an unbalanced load.

Asynchronous partitioned data algorithms read a non-overlapping partition of the database on each computer and mine independently using local minimum support thresholds. One computer then collects all of the locally frequent results from the other computers. Using the idea from Lakshmanan et al.'s Segment Support Map, any result that is locally frequent on one computer is potentially globally frequent, so each computer must rescan its database partition and count all potentially frequent results. These algorithms do not require any communication or barriers during the mining on each computer. Researchers, such as Tanbeer et al., have found that the mining portion of these algorithms scale much better than SPDAs as the number of computers increases because of this. Similar to SPDAs, APDAs reduce the input costs by only reading a partition of the database on each computer.

The problem with APDAs is the cost of re-scanning the database partition on each computer and counting all potentially frequent itemsets or sequences. Re-scanning is necessary since globally frequent itemsets or sequences may not be locally frequent, so the mining process on some computers prunes them, and does not have a readily available count for them. As the number of computers increases, the number of potentially frequent itemsets or sequences may increase as well, since lower local minimum support thresholds may result in more locally frequent sequences. Locally frequent sequences are not necessarily globally frequent, so as the number of locally frequent sequences increases, the cost of re-scanning the database increases as well.

Asynchronous duplicated data algorithms read the entire database on each computer and divide the load by assigning projections to different computers. The basic ADDA structure does not require any communication during the mining process, just the singleton assignments before mining begins, and the results collection when mining completes. One problem with ADDAs is that, as El-Hajj and Zaiane discovered, having each computer read the entire database from a central server takes much longer than reading a non-overlapping partition of the database on each computer. They reduce the impact of this by having each computer read a partition from the central database, and then exchange partitions with all other computers.

Another problem with the basic ADDA structure is load balancing. El-Hajj and Zaiane assigned singletons based on the number of length two database projections they have. This limits the number of computers that their algorithm can use to mine to the number of singleton items. In databases where one singleton takes much longer to mine than others, the algorithm cannot balance the load, because it cannot break a singleton down into sub-tasks. Cong et al. designed two algorithms that break down singleton assignments into sub-tasks using estimation. Their algorithms mine a sample of the database on the controlling computer to get an idea of how long it takes to mine each singleton and each recursive projection. This leaves all other computers idle while the controlling computer mines the sample database. When singletons are sub-divided, Cong et al.'s algorithm can only reduce the number of projections to assign by the global counts, since the counts in the sample dataset do not necessarily contain all globally frequent projections. This method of sub-division results in their algorithm assigning many more projections than a SPDA such as She et al.'s second algorithm.

None of the existing algorithms can mine as they read the input database. This means that none of them has the opportunity to meet my goal of completing the mining process in the necessary stage of reading the input database. They all perform multiple scans of the database using modifications of serial data mining algorithms. In addition, none of the existing algorithms implements methods to take advantage of shared memory multi-processor, multi-core, or multi-threaded computers. To use more than one processor on a single computer, they would have to use Rabenseifner et al.'s Pure MPI model.

2.4 An Ideal Framework for Mining from a Central Database

Based on the strengths and weaknesses of the presented distributed memory algorithms, an ideal algorithm would start by reading a partition of the database on each computer, to minimize the cost of reading from a central database as the SPDA and APDA techniques do. It would mine the database on each computer without requiring any communication between computers, as the APDA and ADDA techniques do. It also should not require communication to perform load balancing and should support heterogeneous computers. Load balancing should not rely on mining a sample of the

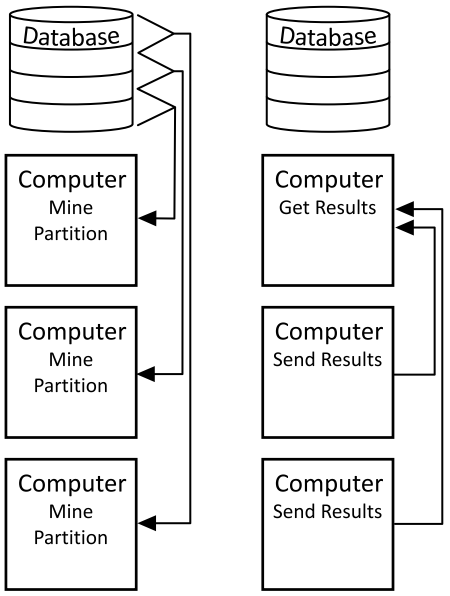


Figure 2.13: Structure of the ideal algorithm

dataset before mining the actual dataset as in Cong et al.'s ADDAs. The only communication should be when the algorithm collects the results, but unlike the ADDAs, it should not require a re-scan of the input database. The ideal algorithm should also provide support for shared memory multi-processor, multi-core, and multi-threaded processors.

Figure 2.13 shows the structure of this ideal algorithm. It simply mines the entire partition on each computer, without communicating during the mining process, and then collects the results.

2.4.1 Requirements of the Ideal Algorithm

To perform the entire mining process in one pass, and not need a second pass to count global frequencies as in the APDAs, the ideal mining algorithm must conform to two main requirements. First off, it cannot capture or transform the database prior to mining. It has to mine directly from the original database as it receives it. Performing a capture or transformation, and then performing multiple mining passes over said capture or transformation, would not achieve the goal of mining in one pass. While algorithms that perform database transformations prior to mining have improved performance on common experimental datasets, Aggarwal et al. [ALWW09] have shown that the performance improvement comes from repeated identical prefixes in the database rows, which overlap in the transformed tree. In their experiments with real world databases, and especially those containing uncertain data, the number of identical prefixes was far lower than found in common experimental databases. The overhead of performing a database transformation when the number of identical prefixes is low, such as done in my research with Leung et al. [LMB08] can outweigh the benefits of said transformation.

Secondly, the ideal mining algorithm cannot apply constraints on a local level that remove itemsets or sequences that could satisfy the constraint on a global level. Minimum support is an example of this type of constraint. The algorithm must apply constraints of this type during or following the result collection stage rather than during the mining process itself. The mining algorithm can apply constraints that only prune itemsets or sequences that do not depend on frequency, such as the succinct constraints I have used along with Leung et al. in our research on uncertain mining [LB09a, LB09b, LHB10, LB10], during the mining process.

The ideal mining algorithm can open many beneficial opportunities by applying all constraints during or after the collection stage. Mining without applying any constraints results in every computer having the count for every possible subsequence or itemset in its database partition available. From that point, if the user wants to adjust the value of a constraint such as minimum support, they can do so without having to mine the database again. In all existing algorithms, if the user wants to lower the minimum support threshold, they have to mine the database again. Itemsets or sequences a higher minimum support threshold pruned may now be frequent. In the ideal algorithm with no constraints during the mining process, it does not prune any itemsets or sequences, so it simply re-scans the results and prints those that satisfy the new constraint.

Similarly, another benefit of not applying constraints during the mining process is, users can add or remove constraints without having to mine the database again. Existing algorithms can only add constraints where the results are a subset of the previous results. Consider the results of mining a web click log using 10% as the minimum support threshold. Users could further constrain this to show only those frequent sequences ending in an error without re-mining using the existing algorithms. With the unconstrained ideal algorithm, users could change the constraints to find all sequences, not just the frequent ones, ending in an error, without having to mine the database again.

Another benefit of not applying constraints during the mining process is that users can add new data and update the results without having to mine the entire database again. In existing algorithms, new data can cause a previously infrequent sequence to become frequent, but since existing algorithms prune infrequent sequences, they have to mine the entire database again. Even in stream mining algorithms, such as my work with Leung et al. [LBY08, LB08] where we used the sliding window approach, the algorithms mine the old data along with the newly received data. The sliding window approach uses a specified number of windows, each holding a set number of database rows, and the algorithm mines the dataset consisting of all windows each time the window containing the oldest data is emptied and refilled. The ideal approach that does not apply constraints during the mining process simply mines the newly added data and adds new sequences to, or increments existing sequence counts in, the old results. It does not have to re-mine previously mined data.

2.5 Summary

In this chapter, I found that it is possible to use both shared- and distributed-memory computers together, and take advantage of the strengths of both, by using MPI and OpenMP. The Hybrid Overlap model is best suited for my frequent-sequence mining algorithm since processors can switch between mining and communicating without waiting for all other processors to complete the current stage. I found that multiple thread units vary in effectiveness depending on the application, specifically, they are less effective in memory bound applications such as data mining, but have a positive effect in most cases. For load balancing, I found that using the runtime of the algorithm is a good prediction of the performance of each computer for future runs of the algorithm, and load balancing algorithms can use this performance measure to assign data.

Next, I discovered that existing distributed-memory frequent-itemset and frequent-sequence mining algorithms follow three different frameworks, namely, synchronous partitioned data algorithms (SPDAs), asynchronous partitioned data algorithms (APDAs), and asynchronous duplicate data algorithms (ADDAs). I found that the algorithms that partition the input data have much lower input cost than those that read the entire dataset on each computer, and the asynchronous algorithms are much more scalable in regards to the number of computers. Existing APDAs have both of these attributes, but require each computer to re-scan its assigned partition once all computers have completed mining and sent their locally frequent sequences to the collecting computer. During the re-scan, each computer counts the number of occurrences of every potentially frequent sequence, since it may have pruned the sequence as locally infrequent during the first scan. Existing research that encounters a performance bottleneck at the central database does not increase the amount of work performed while reading from the database, but rather assumes the database exists locally on each computer before execution.

The goal of my research is to create a frequent-sequence mining algorithm that uses multiple distributed- and shared-memory computers to mine as they read the input database, without using an intermediate structure, and limiting communication so that it is scalable. I created the ideal framework for an algorithm by combining the best attributes of the three existing mining frameworks. The ideal algorithm should be asynchronous and partitioned, but not require a final scan of the database to count potentially frequent sequences. I then created requirements for this ideal algorithm so that it meets my research goals, namely, that it cannot perform multiple scans of the database or use an intermediate structure, and that it cannot apply constraints on a local level that prune sequences that are potentially frequent on the global level.

Finally, I analyzed the benefits of an ideal algorithm meeting my requirements. I found three major benefits, the first being that the user can modify the value of any constraint without having to re-mine the database. Second, the user can add or remove constraints, again without having to re-mine the database. Third, the user can add new data to the database, and add the new results to the old results without having to re-mine any data it has previously mined.

Chapter 3

Single-Scan Sequential Mining Algorithm

In this chapter, I design a single-scan sequential mining algorithm to use heterogeneous distributed memory computers, where each computer may have one or more shared memory processors and thread units, to mine from a central database. The example distributed memory system in Figure 1.1 of my introduction represents this target architecture. I implement my algorithm in C, using MPI to communicate between distributed-memory computers, and OpenMP to provide shared-memory parallelism.

To perform data mining on this type of system, the first step is to have the computers that will perform the mining read the input database. Since this step is necessary, and is a performance bottleneck because all computers share the database, I design my algorithm to perform as much work as possible while reading the database. I limit the amount of communication, and do not require any barriers or synchronizations

Table 3.1: The sub-sequence enumeration of the sequence ⟨ a b c d ⟩

a	b	c	d
⟨ a b ⟩	⟨ a c ⟩	⟨ a d ⟩	⟨ b c ⟩
⟨ b d ⟩	⟨ c d ⟩	⟨ a b c ⟩	⟨ a b d ⟩
⟨ a c d ⟩	⟨ b c d ⟩	⟨ a b c d ⟩	

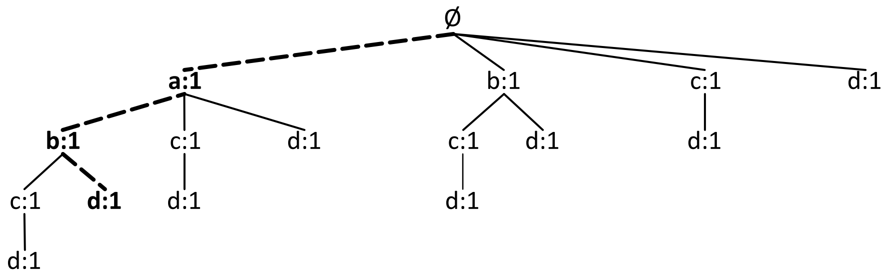


Figure 3.1: The sub-sequence enumeration of the sequence ⟨ a b c d ⟩ as a tree

between all computers, during this process. I also avoid capturing or transforming the initial database so that when my algorithm finishes reading the input database, all that remains for it to perform is the results collection.

My algorithm consists of three components. The first two are the stages from the ideal algorithm diagram in Figure 2.13 of the related work. The first stage, and first component of my algorithm, is using distributed computers to mine the input database. Each of these computers may have multiple processors, cores, or thread units, and as such, should make use of shared memory parallel computing. The second stage, and second component of my algorithm, is collecting the results onto the controlling computer so that the user can view them. The final component of my algorithm is the method I use to balance the load between all computers.

3.1 Mining: Enumeration of Sub-Sequences

For the mining process itself, I enumerate all of the sub-sequences from each row in the database, and store the resulting counts in a tree-based structure. This tree-based structure is not a database capture, but is the results of the mining process. Consider the sequence $\langle$ a b c d $\rangle$ and its sub-sequences, as listed in Table 3.1. All of these sequences are stored in the tree in Figure 3.1. Each node of the tree represents the sequence from the root to that node, followed by the number of times that sequence occurs. For example, the highlighted path in the tree represents the sequence $\langle$ a b d $\rangle$ occurring once. Storing results in this structure is compact since sequences share prefixes. It also allows the algorithm to quickly find sequences and increment their count by tracing the sequence's path through the tree, rather than looking it up in a table.

Each computer performs the enumeration process independently on a non-overlapping partition of the input database, since as I found in defining the ideal mining algorithm for my target system architecture, reading non-overlapping partitions has the lowest input cost. Enumeration does not require any communication between computers, which, as the related work has shown, results in the most scalable algorithms in regards to the number of computers used.

Enumerating all sub-sequences satisfies both requirements of the ideal algorithm I presented in Section 2.4.1. Since enumeration does not require any knowledge of previous or future database rows, my algorithm can completely mine database rows as it reads them, and it does not need to perform additional scans of the database or captured portions of it. Enumeration includes every possible sequence in the results, so my algorithm does not prune any sequences during the mining stage, meaning that it will not have to re-scan the database as Asynchronous Partitioned Data Algorithms do.

The challenge of sub-sequence enumeration is, as I have shown in Section 1.4, there are $2^n - 2$ sub-sequences of each input sequence of length n. In the following section, I design two methods of counting these sub-sequences. The first uses a structure I call an *insertion list* to keep track of the tree nodes that my algorithm needs to update as it enumerates sub sequences. The second takes a depth-first approach to updating the tree nodes.

3.1.1 Enumeration of Sub-Sequences with an Insertion List

The goal of my insertion list is to enumerate all sub-sequences while minimizing the number of operations requiring memory access. The insertion list is an array of length 2^{n-1} where n is the length of the sequence my algorithm is inserting. It works based on the observation that in the enumeration tree, each item in the sequence is a child of every item prior to it in the sequence. For instance, in Figure 3.1, item d is a child of the root node ($\varnothing$), a, b, and c, which all occur before it in the input sequence. The insertion list contains pointers to every tree node that the current enumeration has created or incremented, excluding those created or incremented to insert the last item of a sequence, so that my enumeration algorithm can quickly insert an item as a child of every node already in the list.

Algorithm 1 shows the pseudo code for my enumeration algorithm using an insertion list. I pass the insertion list to the enumeration algorithm so that it does not

Algorithm 1 Pseudo-code for enumerating sub-sequences using an insertion list

```
// insList[0] = Tree.rootNode for all calls to this function
EnumerateSubSequence( insList[], sequence[])
    //variables
    updatedNodePtr //pointer to the newly added or updated tree node
    insListLength = 1 //the current length of the insertion list
    curr //the current position list index a child is being added to
    inputIndex = 0 //the index of the next in the input sequence
    insListIndex //the position of the insertion list to insert to

    //enumeration
    //while there are still items to insert
    while inputIndex < sequence.length
        curr = insListLength  1 //insert from the end of the list
        insListLength *= 2 //double the length of the insertion list
        insListIndex = insListLength  1 //insert from the new list end

        //loop until curr reaches the start of the list
        for ; curr >= 0; curr--, insListIndex--
            //insert the item as a child of the current node
            updatedNodePtr = insertChild(positionList[curr],
                                                      sequence[inputIndex])
            //if not done, put the new node in the list
            if inputIndex < sequence.length
                insList[insListIndex] = updatedNodePtr
        endFor
        inputIndex++ //move to the next item from the input sequence
    endWhile
```

have to allocate and deallocate memory for the insertion list during each sequence insertion. The insertion list initially contains a pointer to the root of the tree, and the initial length of the insertion list is 1, since the root node is the only item it contains. The initial index into the input sequence is 0, the first item in the input sequence array.

While there are still items in the input sequence that my algorithm has not inserted, meaning the input index is still less than the length of the input sequence, it inserts the current item as a child of all existing items in the insertion list. It does this in reverse order so that it inserts the child of the longest sequence, which is at the bottom left most node of the tree in Figure 3.1, first, and then moves to the right across the tree. If the item it is adding is not the last item in the sequence, it adds pointers to all inserted tree nodes to the insertion list, starting from the new end of the insertion list. Since the number of inserted nodes doubles each time my algorithm inserts a new item, the length of the insertion list also doubles. This makes the list index to start inserting from one less than double the old insertion list length.

Figure 3.2 shows how my enumeration algorithm creates the tree in Figure 3.1 using an insertion list. The example tree is initially empty, consisting of only a root node. The insertion list contains a pointer to the root node. My enumeration algorithm then reads the first item, a, from the input sequence. It inserts a as a child of the root node, and adds a pointer to a at the end of the insertion list. It then reads the next item, b, from the input sequence, adds it as a child of both nodes pointed to by the insertion list, and adds the newly created nodes to the insertion list. Next, it reads c from the input sequence, and follows the same procedure, inserting c to the tree and pointers to the new nodes to the insertion list. Finally, it reads d from the input sequence and adds it as a child of all nodes in the insertion list. As d is the last item in the input sequence, my algorithm does not add the newly created nodes to the list, since it has completed enumerating the sequence.

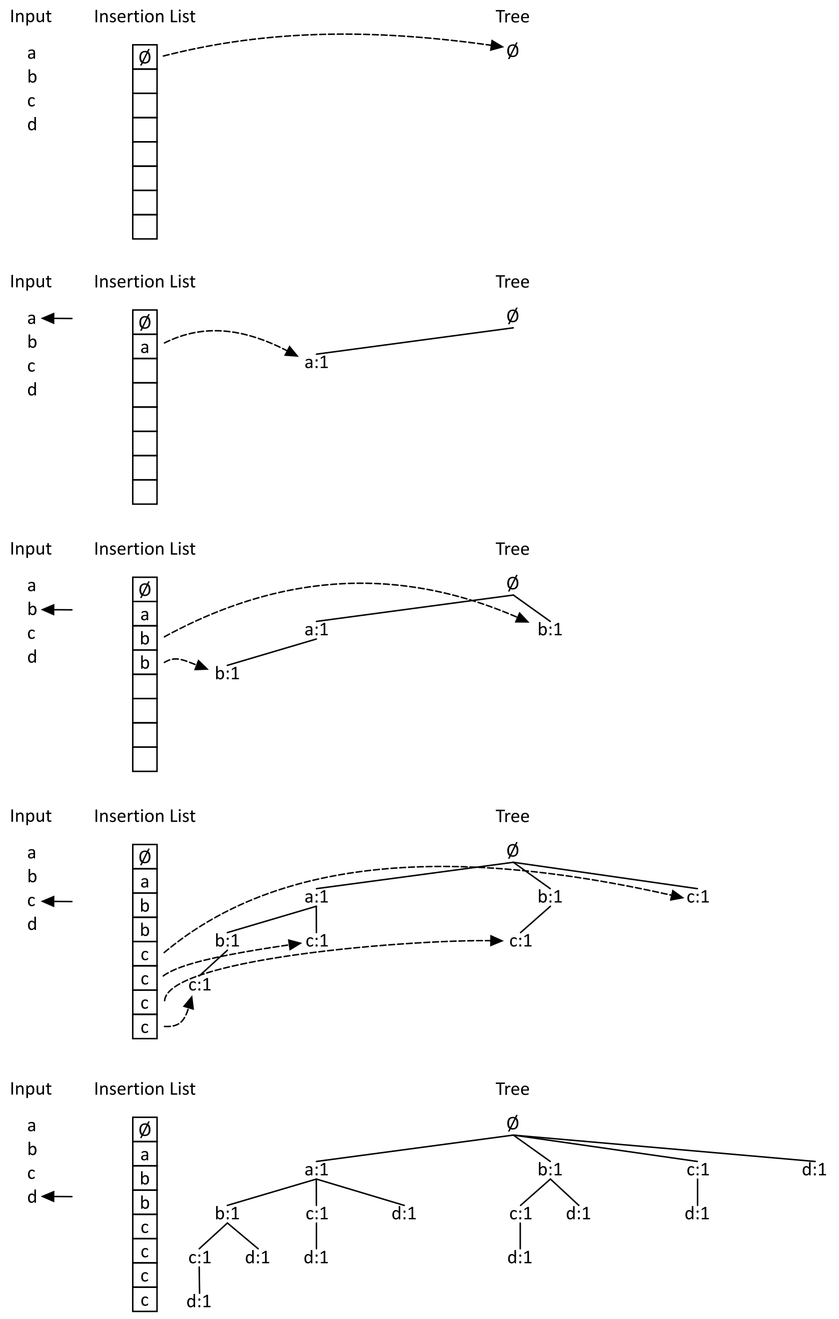


Figure 3.2: Enumerating sub-sequences with an insertion list

Using this method, my algorithm reads each item in the input sequence exactly once. It inserts items as children of every element in the insertion list, which requires reading the parent node and writing the child node. Because the insertion list contains pointers to every node that will be a parent of the item it is inserting, my algorithm never needs to read multiple nodes to find the parent of a new node. This means the algorithm needs $2^n - 1$ node reads and writes to insert all $2^n - 1$ sub-sequences. The insertion list stores pointers to all nodes, except those created by the last item in the input sequence, so my algorithm requires a total of $2^{n-1} - 1$ insertion list writes.

3.1.2 Depth-First Sub-Sequence Enumeration

Since my insertion list structure is large, requiring an array of length 2^{n-1}, I now also create a depth-first sub-sequence enumeration algorithm. The depth-first algorithm keeps a last in, first out stack containing the nodes it has inserted, excluding those containing the last item from the input sequence, as well as the index in the input sequence of the next item to insert as a child of that node. The stack is an array of n items for an input sequence of length n, since at the most, it will contain the root node and the first $n - 1$ items of the input sequence.

Algorithm 2 shows the pseudo code for my enumeration algorithm using a depth-first approach. I pass the node stack and index stack to the enumeration algorithm so that it does not have to allocate and deallocate memory for the stacks during each sequence insertion. The node stack initially contains a pointer to the root node of the tree. At the beginning of each enumeration, my algorithm sets the next input sequence index of the root node to 0, and starts the stack index at 0 as well.

My enumeration algorithm then loops until there are no more entries on the stack. When there is an item on the node stack, my algorithm gets the item from the input sequence at the node's corresponding index stack entry, adds it as a child of the node, and then increments the node's corresponding index stack entry. If the newly inserted item is not the last item in the sequence, my algorithm pushes the newly created node onto the node stack, and sets the corresponding entry in the index stack to the value of the parent nodes corresponding index stack entry. If the newly inserted item is the last item in the input sequence, my algorithm pops nodes off the stack until it finds a node that it still needs to insert children to.

Figure 3.3 shows how my enumeration algorithm creates the tree in Figure 3.1 using the depth-first insertion method. The example starts with an empty tree, consisting of only the root node, and the next item to insert to the root node is the first item in the sequence. My algorithm starts by reading the first item, a, from the input sequence, inserting it as a child of the root node, and incrementing the root node's next child index. Since the newly inserted node, which contains a, is not the last item in the input sequence, my algorithm pushes a pointer to it onto the stack, and sets its next child index. The next child index for a newly inserted node is always the same as the incremented next child index from the parent. My algorithm then adds b as a child of a, and c as a child of b, in the same manner.

When my algorithm inserts the last item in the input sequence, d, it does not push it onto the stack, since it does not have any children. After inserting item d as a child of c, the next index for c is 5, which is not in the input sequence, so my algorithm pops c off the stack. The item at the top of the stack is now b with a next

Algorithm 2 Pseudo-code for enumerating sub-sequences using a depth-first approach

```
// nodeStack[0] = Tree.rootNode for all calls to this function
EnumerateSubSequence(nodeStack[], indexStack[], sequence[])
    //variables
    updatedNodePtr //a pointer to the new or updated node
    stackIndex = 0; //index of the current item on the stack
    indexStack[0] = 0 //initialize the next parent as the root node
    //enumerate
    while stackIndex >= 0 //while there are still items in the stack
        //insert the item as a child of the current node
        updatedNodePtr = insertChild(nodeStack[stackIndex],
                                     sequence[indexStack[stackIndex]])
        //increment the next index for the current stack node
        indexStack[stackIndex]++

        //if the new next index is not the last item
        if indexStack[stackIndex] < sequence.length
            //push the new node onto the stack
            nodeStack[++stackIndex] = updatedNodePtr
            //set the next index to the next input item
            indexStack[stackIndex] =
                                  indexStack[stackIndex - 1]
        else
            //while there are still nodes on the stack
            //and all input items have been inserted
            //as children of the current node
            while stackIndex >= 0 and
                     indexStack[stackIndex] ==
                     sequence.length
                stackIndex-- //pop the completed nodes off the stack
    endWhile
```

index of 4 so my algorithm reads the item at position 4 of the input sequence, d, and inserts it as a child of b, incrementing the next child index as it does so. My algorithm continues this pattern, moving up to a in the stack, inserting c as a child of a, d as a child of c, moving up to a in the stack again, inserting d as a child of a, and so on.

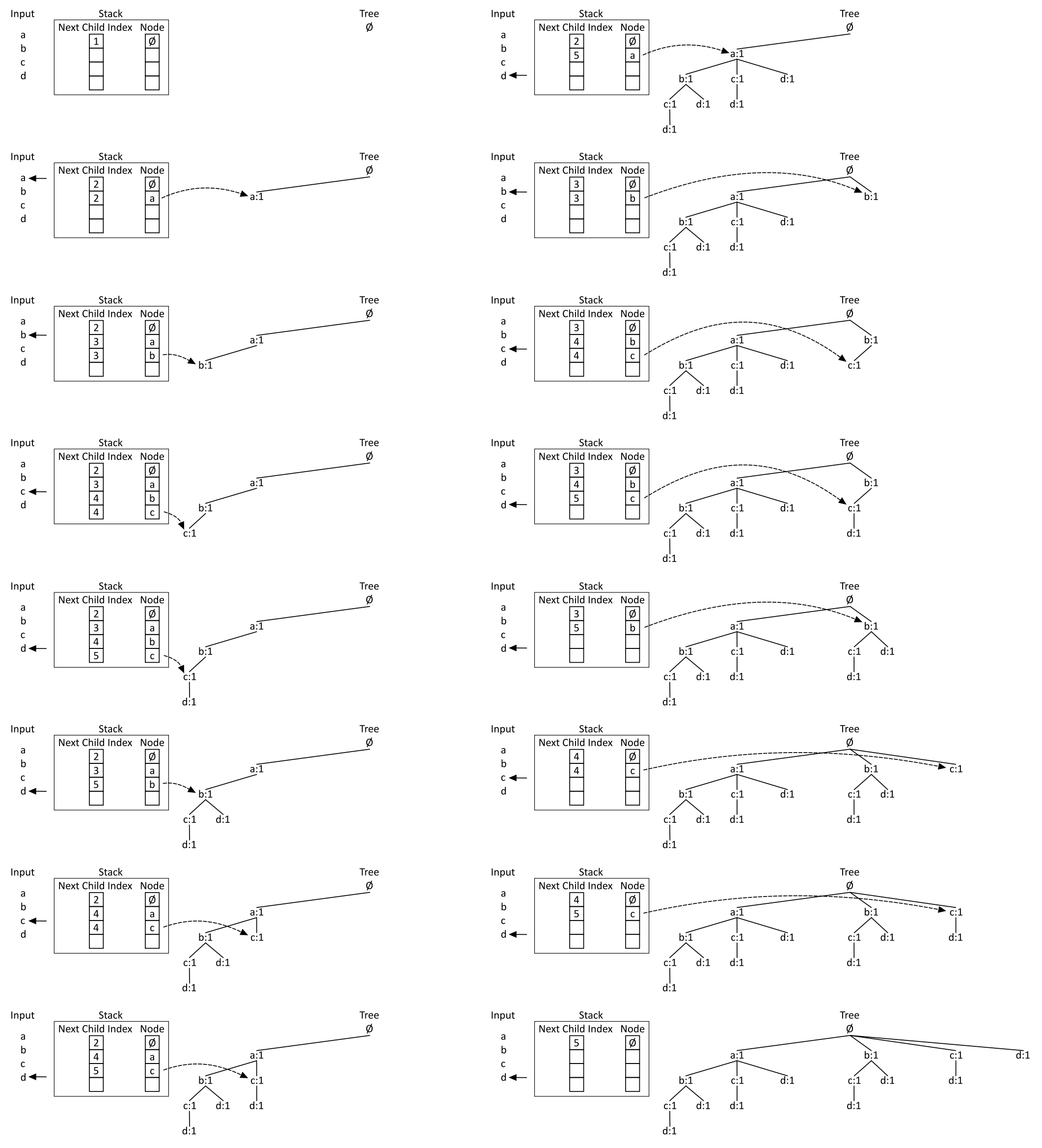


Figure 3.3: Depth-first sub-sequence enumeration

Using this method, unlike my insertion list method, it performs $2^n - 1$ input reads since it has to read the item from the input sequence at each node insertion. Similar to my insertion list method, the depth-first insertion method contains a pointer to the parent of a new item, so it never has to perform a root to parent transversal, and requires one parent read for each node write. It requires $2^n - 1$ node reads and writes. The depth-first insertion method pushes every parent node to the stack once, so it requires $2^{n-1} - 1$ stack writes.

3.1.3 Sub-Sequence Enumeration on Shared Memory Computers

As I have outlined in Section 2.4, one of the requirements of the ideal framework for mining from a central database, and one of the goals of my algorithm, is to make use of shared memory computers. This is important since, as I mentioned in Section 1.1, computers with multiple cores are becoming increasingly common. Another consideration is the benefit of Hyper-Threading enabled processors. Both Liao et al. and Curtis-Maury et al. found that Hyper-Threading increases competition for shared memory, and may cause performance degradation in memory intensive applications. My enumeration algorithm is memory intensive, but also requires input from disk, which will cause threads to stop accessing memory while waiting for data to load from disk.

I use OpenMP to implement shared-memory parallelism in my enumeration algorithm. Both my insertion list and my depth-first sub-sequence enumeration work in parallel in the same way. Each OpenMP process loops over two functions, one to read a sequence from the central database, and one to enumerate sub sequences and store them into the results tree. I share the results tree between all OpenMP processes. The OpenMP processes read sequences from the database using a greedy approach, that is, as soon as a process finishes one enumeration, it reads another one from the input database.

This does not provide any load balancing between OpenMP processes, but even in the worst-case scenario, where all processors complete their current enumerations at the same time, but only one sequence in the database still requires enumeration, the load imbalance is low. Even if the last sequence is long, processors are only idle while the last processor performs one sub-sequence enumeration. Assigning an equal portion of the input sequences to each processor has greater potential for load imbalance. Consider a database where all of the longest input sequences occur together. The processors enumerating the longer sequences would perform much more work than those enumerating the shorter sequences.

Multiple processors can potentially try to load an input sequence from the database at the same time. Since the greedy approach does not partition the input sequences between processors before they begin reading and enumerating, the sequence loading function requires a critical section, so that one processor completes loading a sequence before the next begins. This creates a potential performance bottleneck in the application, as it can potentially leave processors idle waiting for other processors to complete input. Fortunately, if this occurs, it means that reading from the central database is the bottleneck. Forcing the database to be the bottleneck, and completing the mining process in near the time it takes my algorithm to read the input database, is the goal of my algorithm.

For both methods, the enumerate sub sequences function is called independently by each process. Each process requires its own enumeration structure, that is, its own insertion list or node and index stacks. My OpenMP implementation shares the results tree between all processors. This means that all processors can potentially attempt to create or update the same tree node simultaneously. If a process is inserting a new child to a parent node, it has to lock the parent node while it creates the child, to avoid processes creating multiple copies of the child node.

Fortunately, this is the only case in which my OpenMP enumeration algorithm needs to lock a tree node. My enumeration tree stores the children of a node as a list ordered by the items the children contain. If an item does not exist as a child of a node, my algorithm already has a pointer to the last child with an item occurring before the item it is inserting. It locks the parent node at this point and searches from this pointer, rather than repeating the entire search, to see if another processor has created the required child before it locked the node. The processor unlocks the parent node if the required child now exists, or creates the required child and then unlocks the parent if it does not. Algorithm 3 shows the pseudo code to insert a child node. The function to insert children to a locked node is nearly the same, but it does not have to lock or unlock the parent, does not need to check if pointers have changed since it tested and set them since the calling function locked the node, and starts searching from where the insert child function left off.

When my algorithm updates the count of a node, it does not need to lock it, but rather uses an atomic operation to increment the count. Atomic operations are similar to critical sections, but are much lighter weight, ensuring only that no other processes load from or store to the memory referred to by an atomic instruction, until that instruction completes.

Algorithm 3 Pseudo-code of the algorithm to insert child nodes

```
//insert the passed item as a child of the parent node
InsertChild(parentNode, childItem)
    //variables
    updatedNode //the node created or updated by the insertion
    inserted //a flag set when the node has been inserted

    //if the parent has no children,
    //or the new node comes before the first child
    if parentNode.firstChild == NULL or
            parentNode.firstChild.item > childItem
        parentNode.lock() //lock the parent node
        //insert the child to the locked parent node
        //NULL means the insertion search starts from the first child
        updatedNode = InsertChildLocked(parentNode,
                                        NULL, firstChild)
        parentNode.unlock() //unlock the parent node

    //else if the first child is the node to update
    else if parentNode.firstChild.item == childItem
        //set the updated node to the first child
        updatedNode = parentNode.firstChild

        //since the parent is not locked, the first child could have
        //changed between the check and the set, so make sure
        //the updatedNode is the correct node
        while updatedNode.item != childItem
            //move to the next node
            updatedNode = updatedNode.sibling

    //else the item to insert or update is not the first child
    else
        //use updatedNode to search the list
        updatedNode = parent.firstChild
        inserted = false //the item has not been inserted yet

        while not inserted //loop until the inserted flag is true
            //if all siblings have been checked or the item in the
            //next sibling comes after the item being inserted
            if updatedNode.sibling == NULL or
                    updatedNode.sibling.item > childItem
```

```
//InsertChild continued
                parentNode.lock() //lock the parent node
                //insert the node after the last node with an item
                //that comes before the item being inserted
                updatedNode = insertChildLocked(parentNode,
                                                updatedNode, childItem)
                parentNode.unlock() //unlock the parent node
                inserted = true //set the flag to end the loop

            //else if the first sibling is the node to update
            else if updatedNode.sibling.item == item
                //set it as the updated node
                updatedNode = updatedNode.sibling

                //since the parent node is not locked, the next
                //sibling could have changed between the check
                //and set, so ensure it is correct
                while updatedNode.item != childItem
                    //move to the next node
                    updatedNode = updatedNode.sibling

                inserted = true //set the flag to end the loop

            //else the next sibling's item occurs before the
            //childItem, so move across the siblings list
            else
                //move to the next sibling
                updatedNode = updatedNode.sibling
        endWhile
    endElse

    //atomic update of the nodes count
    updatedNode.count++
```

3.1.4 Summary on Mining or Enumeration of Sub-Sequences

My depth-first and Insertion list approaches to sub-sequence enumeration both require the same amount of node reads and writes to build the enumeration tree.

They also both store the same number of nodes into their insertion data structures. The difference between the two structures is that, for an input sequence of length n, the depth-first approach requires $2^n - 1$ reads of the input sequence, while the insertion list approach only requires n reads of the input sequence. The trade-off is that, while the depth-first approach requires $2n$ array entries for the node and index stacks, the insertion list requires 2^{n-1} array entries. Both enumeration methods achieve all of the benefits of the ideal mining framework I presented in Section 2.4.1. The user can modify constraint values, add or remove constraints, and add new data, without having to re-run the mining algorithm on data it has previously mined.

My shared memory implementation allows all local processors to update the same enumeration tree. Each processor has its own enumerating structure, be it an insertion list or node and index stacks, and performs enumeration with very little interaction between processors. The only locking required is locking the parent node when a processor creates a new node. Since both of my insertion algorithms only create each node once, and they search the child list for nodes prior to locking the parent, the impact of these locks is low. There is a critical section when loading input from the database but it only reads a row from the database and stores it in array before exiting. If this critical section limits performance, it means that the central database cannot keep up with the mining computers, and has become the performance bottleneck. This is the goal of my research; I want to complete the mining process in as close to the time it takes to read the central database as possible.

3.2 Result Collection

Since one focus of my algorithm is avoiding barriers where all computers must complete a task before any of them can move on, I have chosen to use Rabenseifner et al.'s Hybrid Overlap model, which allows shared memory processors to perform MPI communication within OpenMP parallel sections. Each distributed computer has a partition of the input database assigned to it, and then independently enumerates all sub-sequences, so the mining process itself does not require any communication. The only communication required is for results collection.

In the following section, I create two methods for collecting results. The first, using the idea of Lakshmanan et al.'s Segment Support Map, is to have computers send sequences to the collecting computer to flag them as potentially globally frequent as soon as they become locally frequent. Once all computers have completed mining, the collecting computer only needs to collect the potentially globally frequent sequences. I call this approach collecting potentially frequent sequences. My second method is to have each computer serialize its entire local enumeration tree and send it to the collecting computer as soon as it completes the enumeration process. I call this second approach collecting all sequences.

3.2.1 Collection of Potentially Frequent Sequences

My first approach to results collection, sending locally frequent sequences to the root node as they become frequent, requires a modification to the enumeration algorithm. Both of my enumeration methods insert the last item in the input sequence as a child of every other node created by the enumeration. Because of this, both methods check if a sequence has become locally frequent when inserting the last item from the input sequence. If the node containing the last input item is locally frequent, but the algorithm has not flagged it as sent, it just became locally frequent. My algorithm then sends the root to node path to the collecting computer to flag the sequence. The collecting computer flags the sequence as potentially globally frequent in its local results tree, inserting nodes with a count of zero if they do not exist. If the node containing the last input item is not locally frequent, my algorithm checks if the parent is newly locally frequent, and sends the root to parent path to the collecting computer if it is.

Communication in this stage is between the flagging computer and the collecting computer only. It does not require a barrier between all computers before sending a sequence to flag. Using the Hybrid Overlap model, one process on the collecting computer listens for incoming sequences, while the rest of the processes mine.

Using this technique, only flagging sequences when inserting the last child and checking the last child before the parent, reduces the amount of sequences that my algorithm needs to send. Since the root to parent path is a prefix of the root to child path, sending the root to child path flags the entire root to parent path, so only one send is necessary when both parent and child become frequent during the same insertion. Both of my enumeration methods start node insertions from the longest path in the tree, the depth-first approach by definition, and the insertion list approach by inserting children starting from the end of the list. This, combined with only checking for new locally frequent nodes when inserting the last input item, ensures that my algorithm will never separately send a sequence and a prefix of the sequence that both became frequent during the same input sequence enumeration.

Flagging Results

When a node becomes frequent, my algorithm performs a root to node trace to get the newly frequent sequence, which is the root to newly frequent node path in the tree. It then sends the sequence to the collecting computer to flag. Using my depth-first enumeration method, this trace is simple, since the node stack contains the current path through the tree. If the updated child node is frequent, the new sequence is the items from the node stack, and the item from the child node. If the updated child node is not frequent, but the parent node became frequent, the sequence is the items from the node stack without the item from the updated child node.

Using my insertion list to enumerate sub-sequences makes this sequence trace more complicated. The trace algorithm needs a way to calculate which insertion list entry is the parent of a node. My insertion list algorithm knows the insertion list index, that is, the position in the insertion list, for any item that it inserts to the tree, even those that it does not have to add to the list. It can trace the path from a node to the root using this index. I base my tracing method on the way I fill the insertion list. Each item insert doubles the size of the list, and I add items from the end to the start of the new space, as children of the nodes from the end to the start of the list. The result of this is each power of two in the list is the start of a new item. The number of list entries for an item is the same as the number of all list entries that have come before it, and I insert all list entries in the same order. Finding the

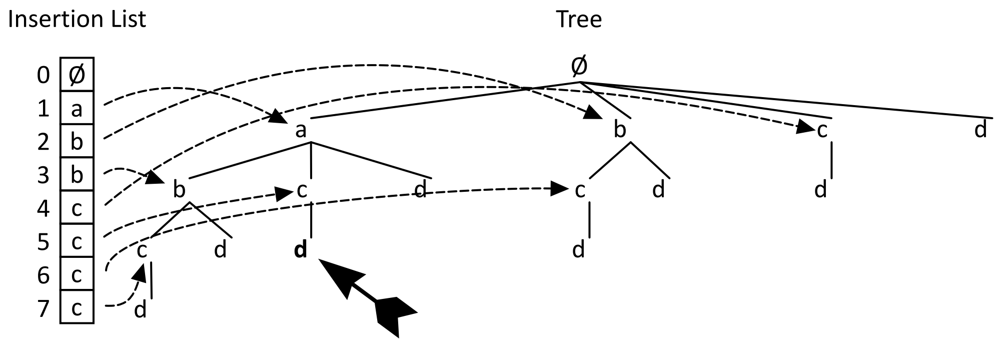


Figure 3.4: Tracing a node to parent path using an insertion list

power of two that an item starts at, and then subtracting it from the current node's insertion list index, gives the index of the parent node.

For example, consider the case where the highlighted node, d, becomes frequent in Figure 3.4. The insertion list index for d is 13, since my insertion algorithm starts inserting from position 15 in the insertion list, and the highlighted node d is the third item it inserts. The largest power of 2 less than 13 is $2^{4-1} = 8$, so this the d's start position in the insertion list. The index of d's parent in the insertion list is d's index, 13, minus its start position, 8, giving $13 - 8 = 5$ for the parent index. Repeating the process, the parent node for c is $5 - 4 = 1$ and there is no non-negative integer power of two less than 1, so the trace is complete. The traced list indices for the sequence are 1, 5, 13, which correspond to the items a, c, d, in the tree.

Collecting Results

The controlling computer has every node that is locally frequent on one computer flagged once all computers have completed enumerating sub-sequences. It then starts a depth-first transversal of its local tree, visiting only those nodes that it has flagged as potentially being globally frequent. Every time the transversal moves to a potentially frequent child node, it sends a move to child and count request to all other computers. The computers move their local transversals to the same position the collecting computer is in and respond with the count of the requested node. If the collected node turns out to be globally infrequent, the depth-first transversal does not need to search its children, even if the algorithm previously flagged them as potentially frequent. The frequency of a child node cannot be greater than that of the parent since the root to parent path is a sub-sequence of the root to child path. When a node turns out to be globally infrequent, the controlling computer sends a move to sibling and count request.

I have implemented the depth-first results collection as an iterator over all frequent sequences. It consists of three functions. The first function initializes the depth-first transversal on the collecting computer, setting it up so that the iterator stack contains the longest frequent sequence following the first frequent child nodes. The second function also runs on the collecting computer, returning the current frequent sequence from the iterator stack, and setting the stack to contain the next frequent sequence from the depth-first transversal. The third function runs on all other computers, listening for movement requests from the controlling computer, and sending it the requested counts. The respective pseudo code for these three functions is in Algorithm 4, Algorithm 5, and Algorithm 6.

Figure 3.5 shows the stages of my method of collecting only potentially frequent items. During the first stage, each computer reads a non-overlapping partition of the database, and enumerates all sub-sequences from each database row. As soon as a sequence becomes locally frequent on a computer, my algorithm sends the sequence

Algorithm 4 Pseudo-code for initializing the iterator

```
//stack[] -- the stack used for the depth-first search
//length -- the current length of the stack
InitializeIterator(tree, stack[],length)
    length = 0 //set the length to zero
    curr = tree.root.firstChild //the current node being counted
    signal //the signal to send to remote modes

    //find the first potentially frequent child of the root node
    while curr != NULL and curr.flag == FALSE
        curr = curr.sibling
        //set the signal to move to sibling and count
        signal = moveToSiblingAndCount

        //while there is at least one potentially frequent node
        while curr != NULL
            //send the move and count request
            sendRequest(curr.item, signal, length +1)
            //perform a reduction operation to get the global count
            curr.count = MPI_Reduce()

            if curr.count >= minSup //if the new count is frequent
                //push the counted node onto the stack
                stack[length ++] = curr
                //set the new current node to the first child
                curr = curr.firstChild
                //set the signal to move to child and count
                signal = moveToChildAndCount
            else //the new count is not frequent
                curr = curr.sibling //move curr to the next sibling
                //set signal to move to sibling and count
                signal = moveToSiblingAndCount

            //find the next potentially frequent node
            while curr != NULL and curr.flag == FALSE
                curr = curr.sibling
        endWhile
    endWhile
```

Algorithm 5 Pseudo-code for getting the next frequent sequence from the iterator

```
//stack[] -- the stack used for the depth-first search
//length -- the current index of the stack
//sequence  the returned frequent sequence
NextFrequentSequence(tree, stack[],length, sequence)
    curr //the current node being counted
    signal //the signal to send to remote nodes
    sequence //the sequence to return

    if index > 0 //if there is an item on the stack
        copyArray(stack, sequence) //copy stack into sequence
        //pop the last frequent item off the stack
        curr = stack[--length]
        //move to the next sibling of the last frequent item
        curr = curr.sibling

        //find the first potentially frequent sibling
        while curr != NULL and curr.flag == FALSE
            curr = curr.sibling

        //set the signal to move to sibling and count
        signal = moveToSiblingAndCount

        //while there is a potentially frequent node
        while curr != NULL
            //send the move and count request
            sendRequest(curr.item, signal, length+1)
            //perform a reduction operation to get the global count
            curr.count = MPI_Reduce()

            if curr.count >= minSup //if the node is globally frequent
                stack[length++] = curr //push it onto the stack
                //move curr to the first child of the frequent node
                curr = curr.firstChild
                //set the signal to move to child and count
                signal = moveToChildAndCount
            else //the new node is not frequent
                curr = curr.sibling //move to the next sibling
                //set the signal to move to sibling and count
                signal = moveToSiblingAndCount
```

```
//NextFrequentSequence continued
            //find the next potentially frequent node
            while curr != NULL and curr.flag == FALSE
                curr = curr.sibling
        endWhile

        //the stack has been updated to contain
        //the next frequent sequence
        return TRUE
    else
        //there are no more items on the stack,
        //so the transversal is done
        sendRequest(NULL, quit, NULL)
        return FALSE
    endIf
```

to the collecting computer, which flags it as potentially globally frequent. Since each computer is using the Hybrid Overlap MPI plus OpenMP approach, multiple processors on each computer may simultaneously send sequences to the collecting computer. My algorithm waits until all computers have finished mining, since the collecting computer needs to have all locally frequent sequences flagged, before it begins results collection.

The results collection algorithm loops over two stages, sending and receiving movement requests, and sending and receiving counts. There is a barrier after each count reception since the collecting computer must determine if a node is globally frequent before moving to the next node. My algorithm may need to repeat these two stages and the barrier numerous times. With a domain of n items, and a maximum transaction length of l, there can be as many as $n!/(n-l)!$ nodes in the tree. The $n!$ represents the n possible children of the root node, the $n-1$ possible children of each of these nodes, and so on, until it covers every possible permutation of the n domain

Algorithm 6 Pseudo-code for listening to requests on remote computers

```
IteratorListener(tree)
    done = FALSE //flag set to true when a quit signal is received
    length = 0 //the current length of the iterator stack
    inSync = FALSE //if the local and remote computers are in sync
    stack[] //the local depth-first transversal stack
    remoteItem //the item received from the collecting computer
    signal //the signal received from the collecting computer
    remoteLength //the current depth of the collecting computer
    localCount = 0 //the count of the requested node locally

    //push the root node of the tree onto the stack
    stack[length++] = tree.root.firstChild
    while done != TRUE
        (remoteItem, signal, remoteLength) = receiveRequest()

        //if the remote search has moved up the tree
        if length > remoteLength
            length = remoteLength //move up the tree with it

        if signal == quit //if a quit signal was received
            done = TRUE //set the done flag to true
        //if a move to sibling and count signal was received
        else if signal == moveToSiblingAndCount
            //if we're on the same level as the remote transversal
            if length == remoteLength
                //while we haven't found the item
                while stack[length-1].item < remoteItem
                    //move to the next sibling
                    stack[length-1] = stack[length-1].sibling

                //if we found the remote item
                if stack[length-1].item == remoteItem
                    //we are in sync with the collecting computer
                    inSync = TRUE
                    //set the count to send
                    localCount = stack[length-1].count
                else //we didn't find the remote item
                    //we're not in sync with the collecting computer
                    inSync = FALSE
                    localCount = 0 //the local count is zero
            else //the collecting computer is deeper in the tree
                //the local tree does not have the path being collected
                inSync = FALSE
                localCount = 0 //the local count is zero
            endIf

            //send the local count to the collecting computer
            MPI_Reduce(localCount)
        //if a move to child and count signal was received
        else if signal == moveToChildAndCount
            //if we're in sync with the collecting computer
            if inSync == TRUE
                //push the first child onto the stack
                stack[length] = stack[length-1].firstChild
                length++

                //while we haven't found the item
                while stack[length-1].item < remoteItem
                    //move to the next sibling
                    stack[length-1] = stack[length-1].sibling

                //if we found the remote item
                if stack[length-1] == remoteItem
                    //we're in sync with the collecting computer
                    inSync == TRUE
                    //set the count to send
                    localCount = stack[length-1].count
                else //we didn't find the remote item
                    inSync = FALSE
                    localCount = 0 //the local count is zero
            else //else we're out of sync
                localCount = 0 //the local count is zero

            //send the local count to the collecting computer
            MPI_Reduce(localCount)
        endif
    endWhile
```

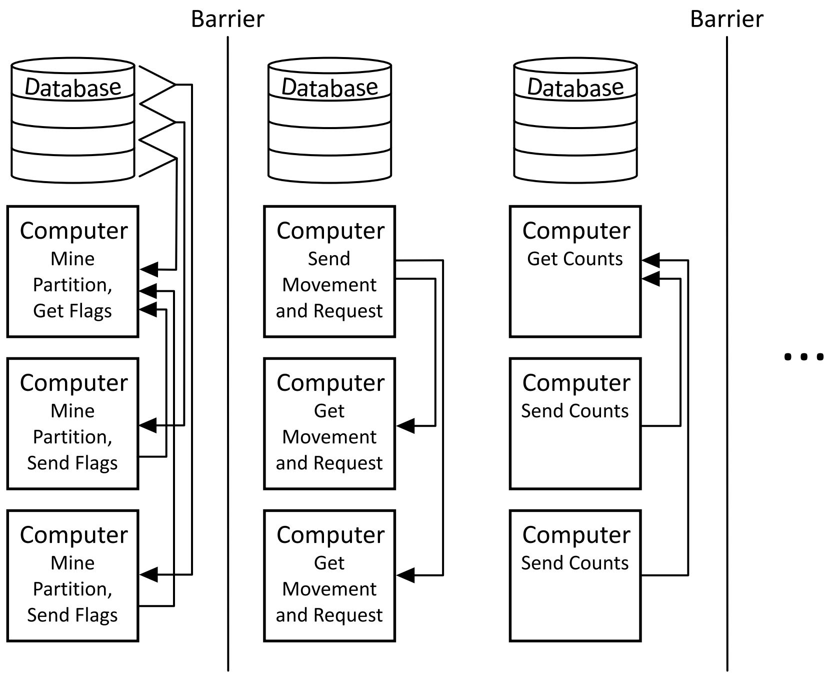

Barrier
Barrier
Database
Database
Database
Computer Mine Partition, Get Flags
Computer Mine Partition, Send Flags
Computer Mine Partition, Send Flags
Computer Send Movement and Request
Computer Get Movement and Request
Computer Get Movement and Request
Computer Get Counts
Computer Send Counts
Computer Send Counts
...

Figure 3.5: Stages of my potentially frequent sequence collection algorithm

items. The $(n-l)!$ represents the $n-l$ nodes that would be children of the nodes on the l^{th} level of the tree, and all of their descendants.

This method of only collecting potentially frequent sequences eliminates the communication required to send nodes that cannot be globally frequent. It meets the goals of the ideal mining algorithm I presented in Section 2.4 using multiple processors and multiple computers. It does not capture or transform the database prior to mining, meaning it mines the data as it reads it from the input database, and does not apply any constraints at the local level that could prune a sequence that is frequent at the global level. It does not have any barriers during the mining stage, since all computers enumerate sub-sequences independently, and the only communication required during this stage is the flagging of potentially globally frequent sequences. These flag sends can occur at most once per tree-node.

The problems with this implementation are that it requires communication, even though the communication does not require barriers, during the mining stage, and requires barriers after every node count in the results collection stage. If the network is the performance bottleneck of the algorithm, increasing the network traffic by flagging locally frequent sequences will decrease network performance, and not allow the central database to become the bottleneck. The barriers required while iterating over the results tree force the algorithm to wait for the slowest computer at each stage. On the other hand, if the network is not the bottleneck, flagging potentially frequent sequences during the mining stage will have little performance impact. Having the algorithm collect only those nodes that it has flagged potentially globally frequent reduces the amount of nodes that it must transfer during the results collection stage. The goal of my algorithm is to mine a database and collect the results in as close to the time it takes to read the database as possible. Reducing the amount of time my algorithm spends collecting results after reading the database allows it to come closer to this goal.

3.2.2 Collection of All Sequences

My second approach to results collection is to have each computer serialize its entire local tree and send it to the collecting computer. Unlike my method of collecting only potentially globally frequent sequences, collecting all sequences does not require any modification of my enumeration algorithms. Each computer enumerates sub-sequences from all of the rows in its database partition independently. When a computer completes sub-sequence enumeration, using either of my enumeration methods, it does not need to wait for any other computers to complete mining; it can start serializing and sending its results. Under the Hybrid Overlap approach of combining MPI with OpenMP, as processors on the collecting computer complete enumerating their last input sequence, they switch from mining to loading serialized trees from other computers.

The collecting computer may not have a processor available to receive a serialized tree directly. To get around this without having to wait for a processor to become available, I have each computer store its serialized tree to disk. If possible, they store it to disk on the collecting computer so that no network communication is required when the collecting computer has a processor available to load the serialized tree. This depends on the operating system to handle running my enumeration algorithm or tree-loading algorithm while receiving files over the network. The benefit of working with files is that they do not require two-way communication. Where my first collection method requires numerous broadcast and reduction operations to send signals and collect counts, this second approach simply saves a tree by opening a file stream and writing to it until the entire tree is stored, or loads a tree by opening a file stream and reading from it until it reaches the end of the file.

I use a depth-first approach to serialize a tree as a series of integers. There are two cases to store in the serial file. The first is a node and its count, which I store as a signed integer and an unsigned integer. The second case is an instruction to pop nodes off the depth-first stack. I have stored these using a negative integer. To collect the result trees my algorithm generates on each computer, I have implemented two functions, one to serialize a tree and save it to disk, and one to load a serialized tree from disk and insert it to the results tree on the collecting computer.

Algorithm 7 shows my tree serialization algorithm. It uses a depth-first method to serialize the tree. Starting from the first child of the root node, my algorithm writes the node's item and its count to the binary file, and then pushes the node onto the stack. It continues adding the first children in this way until it reaches a node with no child. It begins a loop at this point. It pops nodes off the stack, keeping track of the number of nodes removed, until it finds a node with a sibling. Once it finds a node with a sibling, it writes the number of positions it moved up to the binary file as a negative integer. Next, it writes the sibling and its count to the binary file, and then follows all of the first child pointers, adding all of the encountered nodes and their counts to the file and the stack. It goes back to the top of the loop at this point, and repeats it until the stack is empty. The serialize function only uses one local processor.

My tree-loading algorithm works in a similar manner. It reads a signed integer from the file. If the integer is positive, it is an item, and the algorithm inserts it as a child of the node it most recently added to the stack. It then reads an unsigned integer from the file and increments the count of the node by this value. If it reads a negative integer from the file, it adds it to the current stack position, which pops items off the stack. Algorithm 8 shows my tree-loading algorithm. Multiple processors on the collecting computer can run this algorithm simultaneously, using the insert child function I created in Section 3.1.3.

This method requires far less communication during the mining process than my algorithm that collects only potentially frequent nodes. The only communication

Algorithm 7 Pseudo-code for the serialize tree function

```
//serialize the passed tree and store it to the passed file
Serialize(tree, file)
    stack[] //the stack used to perform a depth-first transversal
    stackPos = 0 //the current position in the stack
    upLevels //the number of items popped off the stack
    curr = tree.root.firstChild //the current node being serialized

    while curr != NULL //while there is still an item to insert
        file.write(curr.item, curr.count) //write the item
        stack[stackPos++] = curr //add the node to the stack
        curr = curr.firstChild //move to the first child

    //all first children have now been added to the file and stack
    while stackPos > 0 //while there are still items on the stack
        curr = stack[--stackPos] //pop the top item off the stack
        upLevels = -1 //set the levels moved up to 1

        //loop until we find a sibling or run out of stack nodes
        while curr.siblings == NULL and stackPos > 0
            curr = stack[--stackPos] //pop a node off the stack
            upLevels-- //move up another level

        if curr.siblings != NULL //if we found a node with a sibling
            file.write(upLevels) //write the number of levels moved up
            curr = curr.siblings //move to the sibling

            while curr != NULL //now follow the first child pointers
                file.write(curr.item, curr.count) //serialize the node
                stack[stackPos++] = curr //put the node on the stack
                curr = curr.firstChild //move to the first child
        endIf
    endWhile
```

is computers telling the collecting computer their serial files are ready for it to load. During the results collection stage, computers can serialize their results independently, and not have to wait for other computers to complete the mining process. There is no need to send requests for counts, so in situations where most tree nodes are frequent,

Algorithm 8 Pseudo-code for the serialized tree loading function

```
//load the passed file and insert it into the local tree
LoadSerialized(tree, file)
    stackPos = 1 //the stack contains the root node
    inItem //the integer read from the file
    stack[] //the stack for the depth-first insertion

    stack[0] = tree.root //put the root node on the stack

    inItem = file.readInt() //read a signed integer from file

    while file.EOF != TRUE //while there is more to read
        if inItem > 0 //if the next integer represents a node
            //read the count from the file, insert it and the item
            //as a child node of the top item on the stack,
            //and push the new child onto the stack
            stack[stackPos] = insertChild(stack[stackPos-1],
                                     inItem, file.readUnsignedInt())
            stackPos++ //increment the stack position
        else //the next integer represents an upward move
            //decrement the stack position by adding a negative int
            stackPos += inItem
    endWhile
```

sending the serialized tree has less overhead than requesting counts for each node. Figure 3.6 shows the stages of this method.

Unfortunately, computers do not make use of multiple processors while serializing the tree. Another drawback of my serialization algorithm is that it serializes and transfers entire trees. When not many tree nodes are frequent, it transfers excess data, in comparison with my first result collection method.

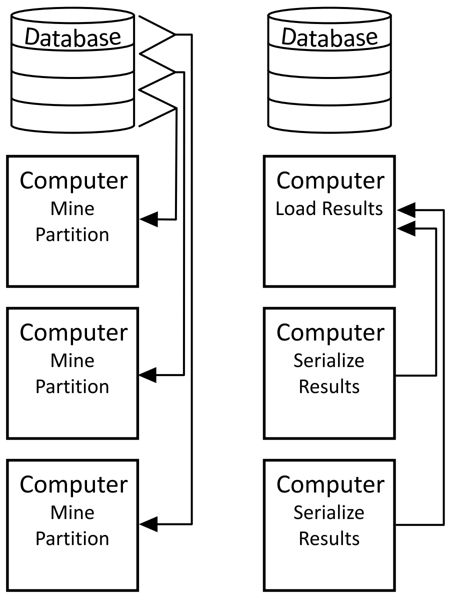


Figure 3.6: Stages of the collecting all sequences method

3.2.3 Summary of Results Collection

I have created two methods for collecting results. The first flags all locally frequent sequences as potentially globally frequent during the mining process, and when the mining process is complete, it only collects those potentially globally frequent sequences. The second serializes the entire local tree on each computer to disk and loads all of these trees on the collecting computer. The first method transfers less tree nodes than the second when there are infrequent tree nodes. It does so at the cost of multiple movement request broadcasts and count reduction operations. The second method transfers the entire local tree from each computer to the collecting computer, but does not require communication during the mining process, and uses a file stream rather than multiple broadcasts and reductions during results collection. Only the second method uses multiple processors during results collection, by having each processor read a serialized file on the collecting computer.

Because my first method performs communication during the mining process, it does not perfectly match the framework of the ideal algorithm. My second method matches the framework for the ideal algorithm. Both algorithms mine a non-overlapping partition of the dataset on each computer, using all available local processors in a shared memory environment, and do not need to transfer projected databases between computers. Neither algorithm requires barriers during the mining process. They both have the potential to complete the mining process in the time it takes to read the input database. The first algorithm only communicates during the mining process to flag potentially frequent sequences, and the second algorithm only sends done signals, so the amount of scalability-limiting communication overhead is low relative to the cost of sub-sequence enumeration.

3.3 Load Balancing

The one requirement of the ideal framework I presented in Section 2.4 but have not yet discussed with either of my enumeration and result collection algorithms is load balancing. The amount of runtime my algorithms require on each computer depends on the size of the database partition it assigns to each computer. As outlined in the ideal framework, the load balancing mechanism should not introduce overhead by requiring communication, and should not prevent algorithms from mining in the time it takes to read from the input database.

As She et al. find with their first Synchronous Partitioned Data Algorithm, and Tanbeer et al. find with their Asynchronous Partitioned Data Algorithm, dividing a large database evenly between homogeneous computers provides a balanced load, and does not incur any load balancing overhead. While sampling the database, such as Cong et al. do in their Asynchronous Duplicated Data Algorithms, can ensure a more balanced load, this type of load balancing causes communication overhead, and further, eliminates the possibility of completing the mining process in the time it takes to read the input database.

To balance load on heterogeneous computers, I follow the approach She et al. and Tanbeer et al. used, and modify the number of rows assigned to each computer without looking at the contents of the rows. Since each computer has different specifications, I use a modification of Galindo et al.'s algorithm to determine the number of rows each computer should receive. As described in Section 2.2, Galindo et al. balance load by starting with the data partitioned evenly between all computers. After a set amount of time has passed, each computer reports how much input it has processed. Their algorithm then reassigns input data, taking some away from the slower performing computers, and moving it to the faster computers. This approach uses the performance of the actual algorithm, with the actual data, as a means to compare the computers and balance the load.

Since my first results collection methods needs to know how many database rows to assign each computer at the start of execution to determine local minimum support, and Galindo et al.'s algorithm introduces overhead, I created an approach that uses the performance of my algorithm, but does not use the actual input data, to compare computers and balance load. I run my distributed mining algorithm with an identical benchmark input dataset assigned to each computer to determine the relative performance of each computer. I use all available processors on each computer, so this load balancing method factors in the overhead from using multiple processors, and I perform result collection to factor in the overhead of my collection methods.

My algorithm partitions the input database relative to performance without any load balancing work during the mining process. Each computer only needs to have its performance measured once. The performance will remain constant unless there are changes to the system. This will not result in a perfectly balanced load, but as I mentioned in Section 2.2, Utrera et al.'s work on load balancing finds that algorithms with some load imbalance but little balancing overhead end up taking less runtime than algorithms that incur high overhead to achieve a perfectly balanced load.

Mining the same dataset on every computer, c, gives the runtime, r_c, for each computer. To determine the percentage of the input database that each computer should read, I first find the reciprocal of each runtime, since my load balancer should assign a larger number of rows to computers with smaller runtimes. I call this reciprocal the score of each computer, and denote it by s_c in Equation (3.1):

$$\frac{1}{r_c} = s_c \tag{3.1}$$

Equation (3.2) shows how to determine p_c, the portion of the input database assigned to each computer, by dividing the score of each computer by the sum of scores from all computers.

$$\frac{s_c}{\sum_{i=1}^{|c|} s_i} = p_c \tag{3.2}$$

Consider this load distribution method running on four computers, with the runtimes from running the benchmark input dataset listed in column Bench Time of Table 3.2. Applying Equation (3.1) gives the scores of each computer, as listed in the Scores column, and applying Equation (3.2) gives the sizes in the Portion column. To show that the portion sizes are correct, consider if the benchmark dataset has 20,000 rows. Dividing the total number of rows by the runtime of the algorithm gives the number of rows that each computer can enumerate per second, with the results in the RpS column. Now consider running the mining algorithm on an input database containing 1,000,000 rows. Multiplying the size of the input database by each computer's portion value gives the number of rows that my algorithm assigns each computer to achieve a balanced load, shown in the Rows column. The Time column shows the result of dividing the number of rows each computer receives by the number of rows per second it can enumerate. Using the portion sizes from the benchmark database, all four computers have the same expected runtime for the 1,000,000 row input database, given the number of rows per second that they can enumerate.

Table 3.2: Sizing partitions based on benchmark database runtimes

Computer	Bench Time	Scores	Portion	RpS	Rows	Time
1	103	0.009708738	0.507322982	194.1747573	507323	2612.71335
2	228	0.004385965	0.229185382	87.71929825	229186	2612.71335
3	303	0.003300330	0.172456327	66.00660066	172456	2612.71335
4	574	0.001742160	0.091035309	34.84320557	91035	2612.71335

To summarize, my load balancing mechanism uses a performance measure of each computer, based on a benchmark run of my algorithm, to partition input data between computers. This method divides the data prior to the execution of my algorithm, so it does not require any communication during execution. Since the number of rows my algorithm assigns to each mining computer does not change, mining computers do not need to use the Hybrid Overlap model to assign a process to receive incoming MPI requests during the mining process, so it can use the maximum available processors for mining.

3.4 Summary

In this chapter, I presented two methods for sub-sequence enumeration, two methods for collecting results, and a method of balancing load between computers. My first sub-sequence enumeration method uses a list of all tree nodes created or updated by the current sequence, so that it only has to read each item from the input sequence once. My second sub-sequence enumeration method takes a depth-first approach, moving through the tree nodes corresponding to the input sequence and inserting or updating children, which requires reading an input item for each node it inserts or updates but uses less memory than the insertion list.

My first results collection method flags sequences as they become locally frequent on any computer, and then collects results from all computers simultaneously, so that it only has to collect those tree nodes that are potentially globally frequent and are children of a globally frequent parent. My second collection mode collects all tree nodes from every computer by serializing them to disk. This lets the collecting computer load serialized trees as soon as they become available and it has a free processor to do so, so it does not require synchronization between computers, but does not reduce the number of collected nodes based on the minimum support.

My load balancing mechanism uses the runtime of a previous run of my algorithm to determine how each computer performs relative to each other. It partitions the data according to this performance so that all computers will complete execution at the same time.

Combining my enumeration methods, results collection methods, and approach to load balancing, gives my new algorithm for mining from a central database in near the time it takes to read the input data. I implement all of the discussed algorithms in C, using MPI for distributed-memory communication, and OpenMP for shared memory parallelism. My algorithms meet the goals I set forth in Section 2.4, the discussion of the ideal mining framework for mining from a central database.

Chapter 4

Experimental Evaluation

In this chapter, I evaluate my algorithm using a series of experiments, and present their results. I implement and test my algorithm, with both insertion and collection modes, in the C programming language [Ker88], using the OpenMP API to utilize shared-memory processors, and the MPI API [Mes] to communicate between distributed-memory computers. I compile and test my code under both Microsoft® Windows® and Scientific Linux® 6 running Linux® Kernel 2.6.32. I used the MPICH [Arg] MPI library on Windows® and the Open MPI [Opea] MPI library on Linux®. Insertion mode 1 is my sub-sequence enumeration algorithm using the insertion list, as I presented in Section 3.1.1, and insertion mode 2 is my algorithm using a depth-first sub-sequence enumeration method, as I presented in Section 3.1.2. Collection mode 1 is my method of collecting only potentially frequent sequences, as I presented in Section 3.2.1, and Collection mode 2 is my method of serializing local trees and loading them on the collecting computer, as I presented in Section 3.2.2.

To measure the performance of different aspects of my algorithm, I now design and run a series of tests. I measure the effects of the number of shared-memory processors, the number of rows in the input dataset, and the length of sequences in the input dataset. I measure the performance of my collection modes as the minimum support threshold decreases, measure the effectiveness of my load balancing mechanism, and test how my algorithm performs as I increase the number of distributed memory computers used. Finally, I compared my algorithm with the PrefixSpan algorithm, using a freely available executable from Illimine [HCC+06]. Before I present my results, I describe the hardware I ran my experiments on, and the input datasets that I used.

4.1 Hardware

I run my experiments on two separate distributed-memory systems. The first is the University of Manitoba Bird Cluster. The Bird Cluster is a homogeneous distributed-memory system running Scientific Linux® 6. A gigabit Ethernet network connects the computers. Each computer has an Intel® Core™ i5-661 processor, which is a dual core processor with Hyper-Threading, and 7.5 gigabytes of DDR3 memory. I store my input databases and results on one of the Data Mining Lab computers, Myna-04, which has a quad-core AMD™ Opteron™ 2360 processor, and is not part of the Bird Cluster.

The second distributed-memory system I use is my heterogeneous cluster of four home computers. The most powerful computer, which mines while hosting the input files and storing the results, has an Intel® Core™ i7-920 processor, which is a quad core processor with Hyper-Threading, 6.0 gigabytes of DDR3 memory, and is running

Microsoft® Windows® 7. The other three computers are an Intel® Core^{TM} i3-530 processor (dual core with Hyper-Threading) with 4.0 gigabytes of DDR3 memory running Windows® Home Server, an Intel® Core^{TM} 2 Quad Q6600 (quad core) with 6.0 gigabytes of memory running Windows® 7, and an Intel® Core^{TM} 2 Duo E5200 (dual core) with 4.0 gigabytes of memory running Windows® Vista. A gigabit Ethernet network connects all four computers.

4.2 Datasets

I use many input datasets to test different aspects of my algorithm. The input datasets are text files. The first line is the number of rows that the file contains. The remaining lines are the rows of the dataset, and consist of a series of tab-delimited integers. The first number is the unique id of the row in the dataset. The second number is the number of items in the row. Finally, the remaining numbers represent the sequence that my algorithm will enumerate. Each integer in the sequences represents the same item across all rows in the dataset. Table 4.1 shows the top left portion of a dataset with ten thousand rows and a row length of five.

Table 4.1: Structure of an input data file

Num Rows/Row ID	Row Length	Item 1	Item 2	...
10000				
1	5	1	5	...
2	5	6	2	...
...	...	...	...	...

The input datasets fall into two categories: repeat, and synthetic. My repeat datasets consist of repeated identical rows. This ensures that every sequence enumeration that my algorithm performs takes the maximum amount of time, requiring the full $2^n - 1$ node insertions. Another effect of the repeat datasets is that, when the load is balanced, counts on all computers will reach the local minimum support threshold at the same time, which is the worst-case scenario for my first insertion mode. This will cause all of the flagging communication to occur at the same time. I have created repeat datasets using ten thousand, one million, ten million, and one hundred million rows, and row lengths of five, ten, fifteen, and twenty. The frequent results of the repeat datasets are all $2^n - 1$ sub-sequences, so I can easily verify the results of my algorithm against these datasets. I have used these repeat datasets to measure the effects of increasing the number of rows, sequence length, number of processors, and number of computers.

I use the IBM® Quest Synthetic Data Generator [AS99], which researchers commonly use to evaluate data-mining algorithms, to create the synthetic dataset. The IBM® Quest group has designed this data generator to create a dataset following realistic supermarket purchase trends. I use it to create a dataset consisting of one million sequences. The average sequence length is 10 items, with a minimum of 0, and a maximum of 20 items. The data generator creates a report with the counts of all sequences it has generated, against which I will verify my results. I use this data set to measure the effectiveness of my load balancing and result-collection methods.

4.3 Experiments and Results

Each of my experiments measures the performance of an aspect of my algorithm, and is the average over three runs. I start my experimentation on one computer, and measure the effectiveness of my shared-memory sequence-enumeration method as the number of local processes increases. Still using one computer, I measure how the performance of my algorithm changes as the number of input sequences increases, and then how it changes as the length of the input sequences increases. Next, I test my results collection methods by mining the synthetic dataset and varying the minimum support threshold. Following this, I evaluate my load balancing technique by comparing the performance of each of my heterogeneous computers. I then measure the performance of my algorithm as the number of distributed-memory computers it uses increases. Finally, I compare my algorithm with the serial PrefixSpan algorithm.

4.3.1 Experiment: Number of Shared-Memory Processors

My first experiment tests the effects of the number of shared-memory processors my algorithm uses. I run my algorithm independently on my Core^{TM} i7-920 computer, which measures the effectiveness of multiple cores and Hyper-Threading, and on my Core^{TM} 2 Q6600 computer, to measure the effectiveness of multiple cores only. I test both of my sub-sequence enumeration algorithms on each computer, and compare their runtime and memory requirements. I perform all of the tests where I vary the number of processors using a repeat dataset consisting of one million transactions containing fifteen items each.

With Hyper-Threading

Figure 4.1 shows the average runtime of my algorithm using insertion mode 1 on my Core^{TM} i7-920 computer, using 1 to 8 processes to utilize the four Hyper-

Threading cores, running on Windows® 7. I predict the time for the next number of processes by multiplying the runtime by the number of processes used, and then dividing by the next number of processes to use. I also include a line for $1/x$, which represents the single process runtime divided by the number of local processes. This line represents the runtime if my algorithm scaled perfectly, that is, if it had no overhead from locks, critical sections, or the OpenMP library itself.

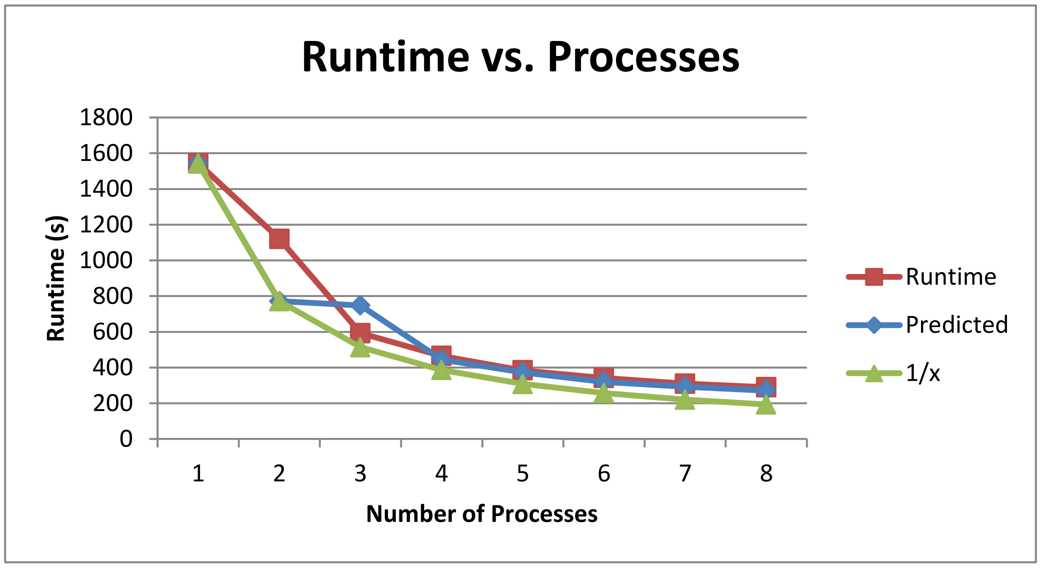


Figure 4.1: Runtime on an i7-920 using insertion mode 1 as the number of processes increases

The average runtime using two processes is well above the predicted and $1/x$ times. This occurs consistently, and seems to be a quirk of Hyper-Threading processors. Past the anomalous two processes time, the decrease in runtime closely matches the predicted and $1/x$ times. The distance between the runtime and $1/x$ times slowly increases, from a difference of 78 seconds using three processes, to 97 seconds using 8 processes. This represents the overhead of OpenMP, the critical section for reading the input, and the cost of locking tree nodes.

Figure 4.2 shows the percentage differences between the runtime and the predicted runtime. The runtime is 45% higher than predicted when two processes are used, but is 21% lower than predicted when three processors are used, bringing it back in line with the $1/x$ time. Past the third process, the differences stay between 3% and 7%. The trend of differences is a low difference for the odd number of processes, followed by a higher difference for the even numbers. This suggests that Windows® 7 is keeping the processes on as few cores as possible, so it is scheduling the odd number processes to new cores, and the even number processes to Hyper-Threading units.

Figure 4.3 shows the increase in memory usage as the number of processors increases. The number of memory allocations increases by six with each processor, representing the extra insertion list for each process, and the extra timers that track runtime. The number of bytes required increases by 65,744 with each additional processor, representing the size of the insertion list, which consists of 2^{14} 4-byte pointers and the memory for the extra timers.

Figures 4.4, 4.5, and 4.6 show the results of the same tests, but using my second insertion method. The average runtime decreases along with the predicted and $1/x$ times until after the fourth process. From the fifth process on, there is a gap between the runtime and the $1/x$ time. Based on the differences between the runtimes and predicted times, it seems that in this case, Windows® is scheduling the first four processes to separate cores, and starts using Hyper-Threading for the fifth process. The runtime using five processes takes an average of 5 seconds longer than the runtime using four, but it decreases as predicted past five processes. The number of memory allocations increases by 9 for each process added, representing the timers and insertion

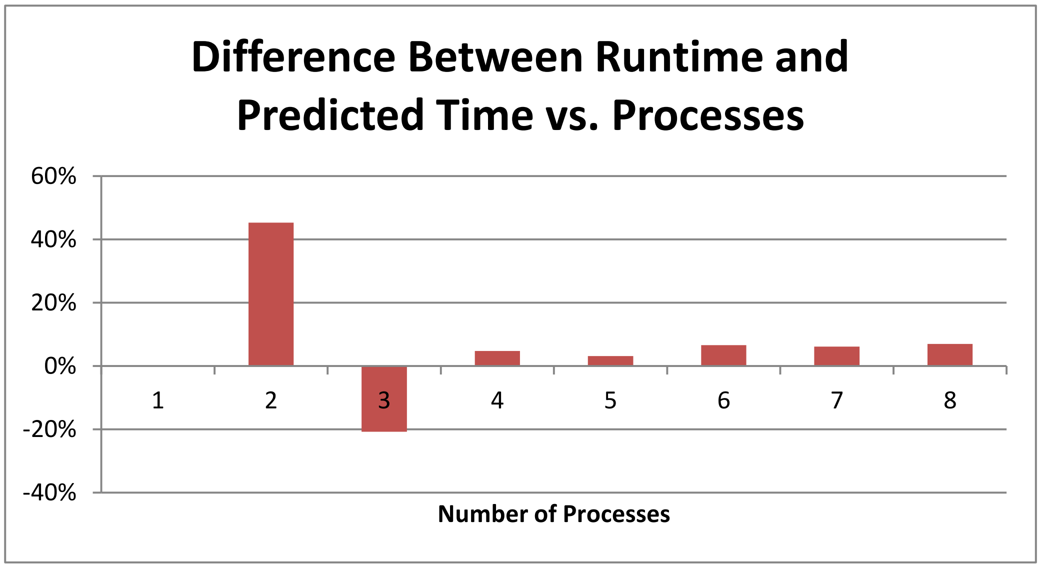


Figure 4.2: Difference between runtime and predicted time on an i7-920 using insertion mode 1 as the number of processes increases

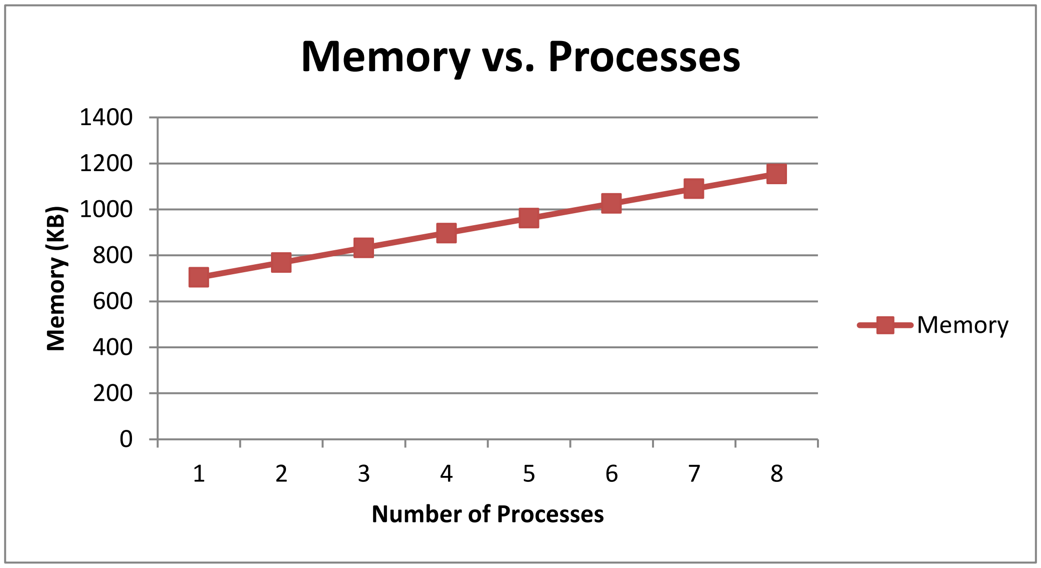


Figure 4.3: The memory usage of insertion mode 1 on an i7-920 using insertion mode 1 as the number of processes increases

stack, and the number of bytes increases by 332, which consists of the two stacks of 14 4-byte pointers and the memory for the extra timers.

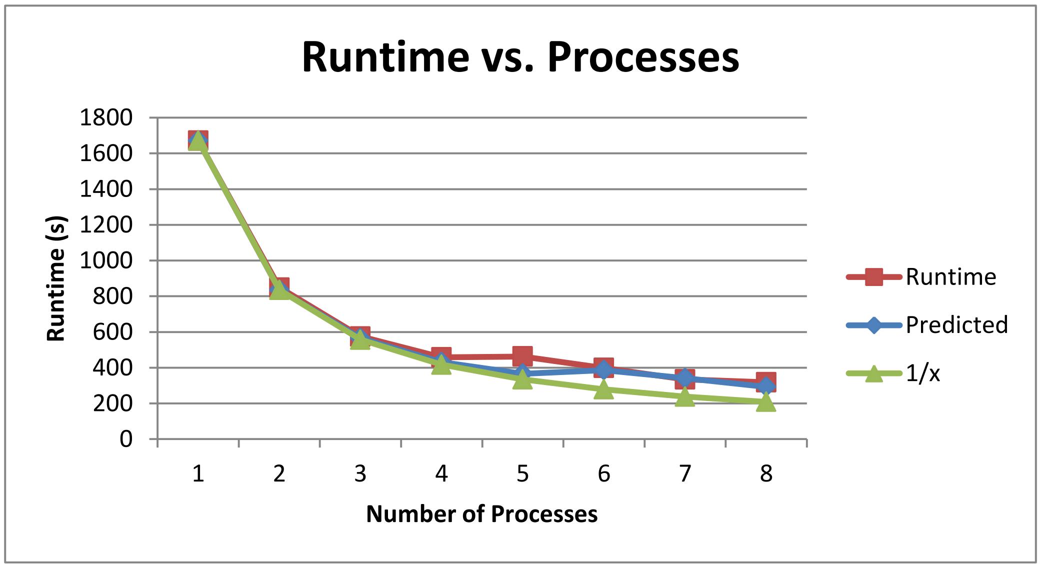


Figure 4.4: Runtime on an i7-920 using insertion mode 2 as the number of processes increases

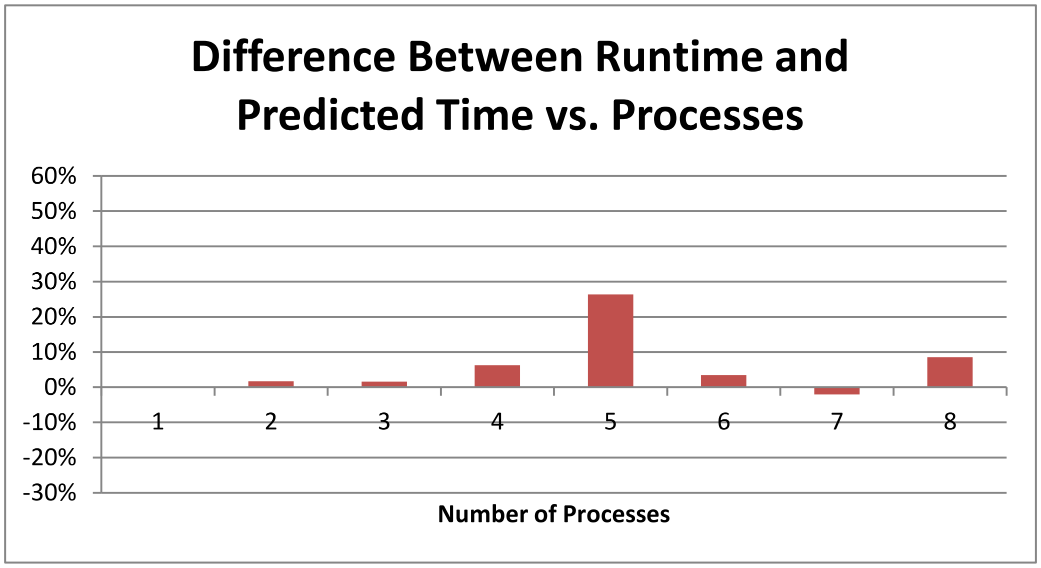


Figure 4.5: Difference between runtime and predicted time on an i7-920 using insertion mode 2 as the number of processes increases

Figures 4.7 and 4.8 compare the runtimes and memory usage of insertion modes 1 and 2. The best performing algorithm switches because of the different scheduling

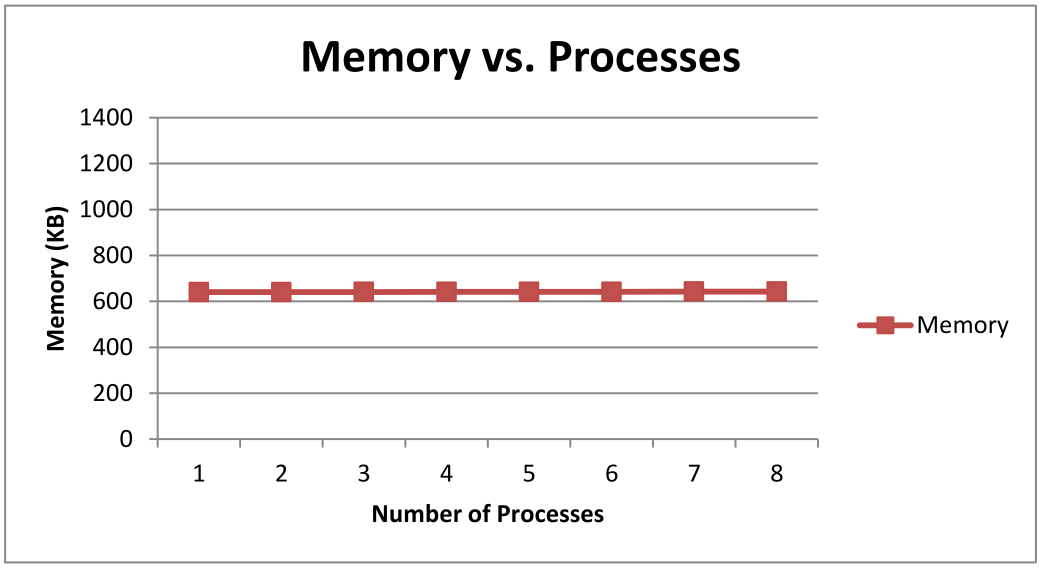


Figure 4.6: The memory usage of insertion mode 1 on an i7-920 using insertion mode 2 as the number of processes increases

strategies used concerning Hyper-Threading. When insertion mode 1 runs with two processes, insertion mode 2 is faster. Insertion mode 2 stays slightly faster with three and four processes, until it starts using Hyper-Threading at 5 processes. Insertion mode 1 is 7% faster than insertion mode 2 with 1 and 7 processes, is 8% faster with 8 processes, and is 16% faster than insertion mode 2 with 5 and 6 processes. The results with 1, 7, and 8 processes are influenced the least by the scheduling differences, giving the most accurate direct performance comparison of the insertion modes. As I have shown, insertion mode 1 requires more memory than insertion mode 2.

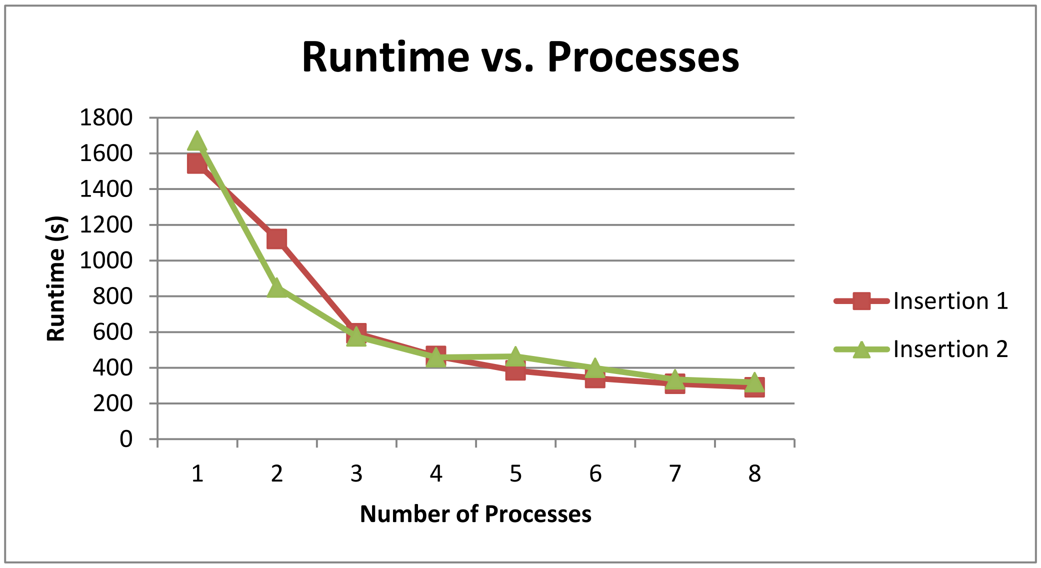


Figure 4.7: Comparison of insertion mode 1 and 2 runtimes on an i7-920 as the number of processes increases

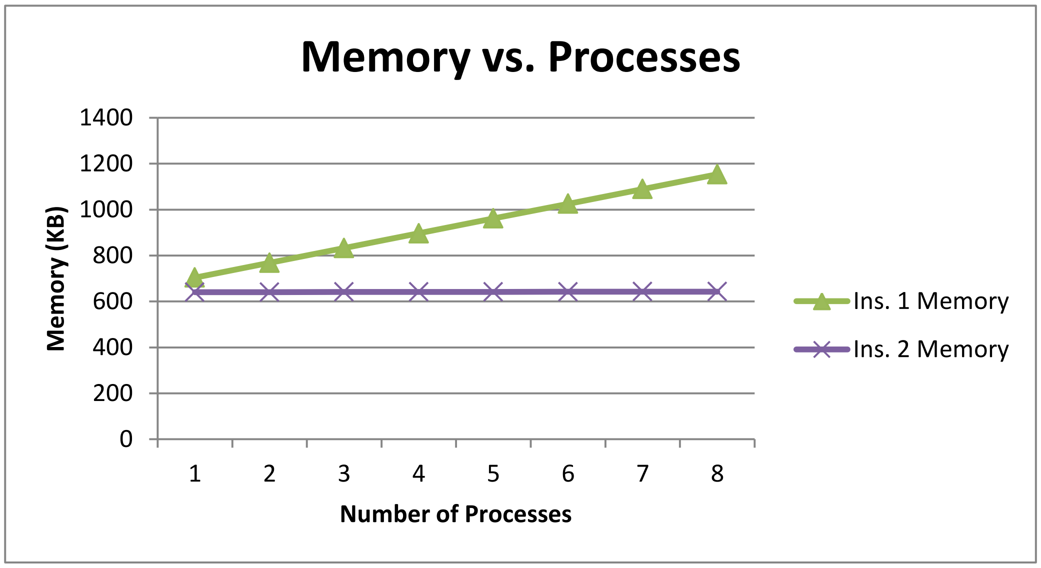


Figure 4.8: Comparison of insertion mode 1 and 2 memory on and i7-920 as the number of processes increases

Without Hyper-Threading

I perform the same tests on both insertion modes on my computer with an Intel® Core^{TM} 2 Q6600 quad core processor to measure the effects of increasing the number of processes on a processor without Hyper-Threading. The memory usage scales the same as it does on the Intel® Core^{TM} i7-920, but the runtime does not have the same anomalies that it does when run on a Hyper-Threading processor. Figure 4.9 shows the average runtime of insertion mode 1 on the Q6600. The runtime decreases following the same trend as the $1/x$ values, and average runtime closes on the predicted runtimes, coming within 5% of it when the number of processes reaches four, as shown in Figure 4.10. The difference between the runtime and the $1/x$ values is 372 seconds when using two processes, and slowly increases, up to 378 seconds with four processes. This shows that the majority of the overhead from OpenMP, locks, and critical sections is incurred when moving to two processes, and that it slowly increases with each additional process.

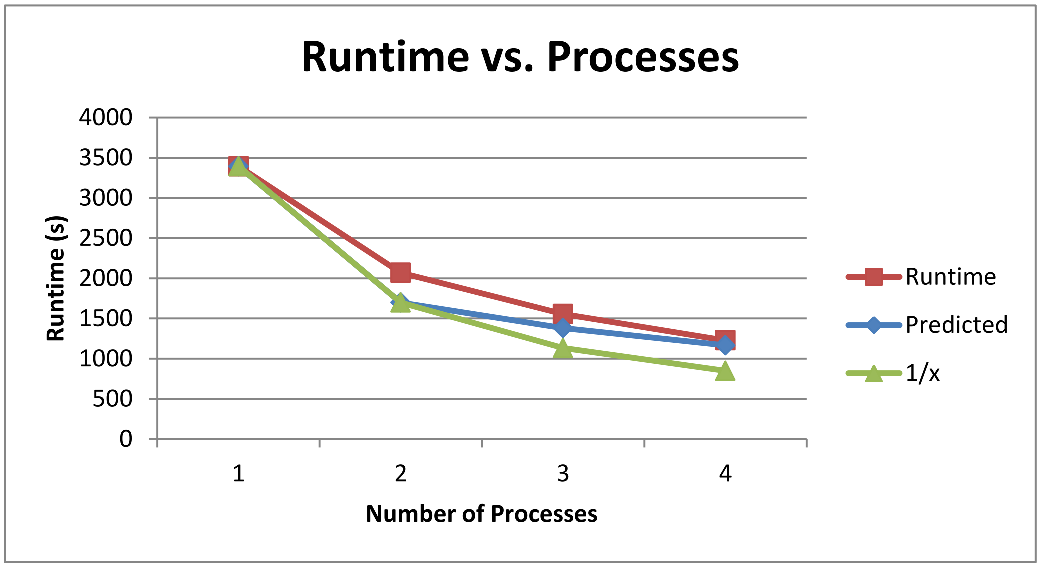


Figure 4.9: Runtime on a Q6600 using insertion mode 1 as the number of processes increases

Using insertion mode 2, the runtimes decrease as the number of cores used increases, again with none of the anomalies seen on the Hyper-Threading processor.

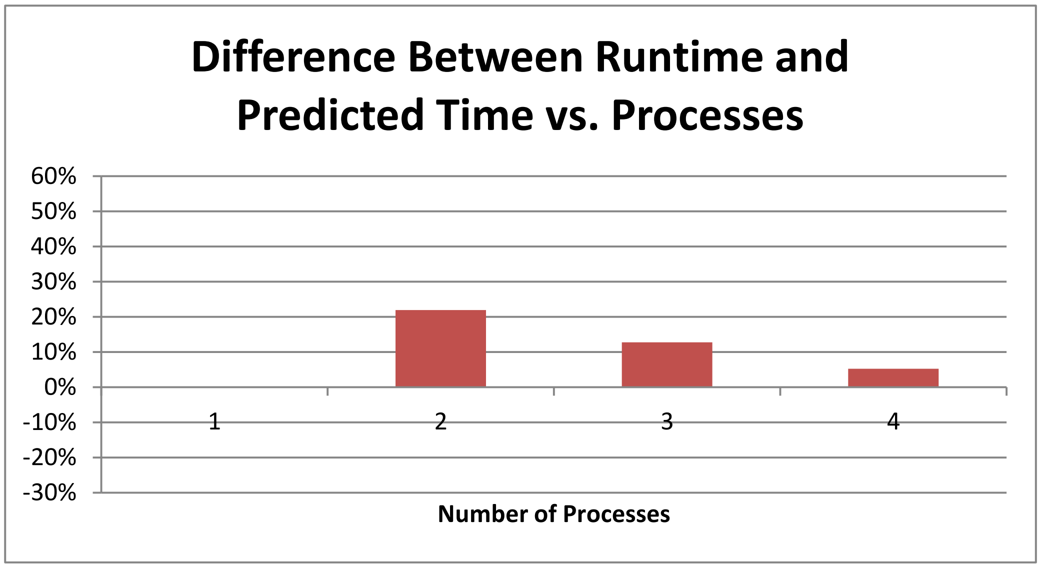


Figure 4.10: Difference between runtime and predicted time on a Q6600 using insertion mode 1 as the number of processes increases

Figure 4.11 shows the average runtime, predicted time, and $1/x$ times of insertion mode 2 on a Q6600. The average runtime and $1/x$ times decrease following the same trend, and the average runtimes converge with the predicted runtimes, coming within 6% of it when the number of processes reaches four. I show this in Figure 4.12. Similar to my first insertion method, the differences between the runtimes and the $1/x$ times slowly increase, moving from 439 seconds with two processes, to 446 seconds with four processes.

Figure 4.13 compares insertion modes 1 and 2 on the Q6600. Unlike the comparison between modes 1 and 2 on the i7-920, the average runtime of insertion mode 1 is always faster than that of insertion mode 2 on this computer, 7% faster with one processor, 8% with two, 9% with three, and 10% with four. The memory required is the same as that for one to four processes on the i7-920.

Considering the results from the i7-920 using 1 and 8 processors, which are not

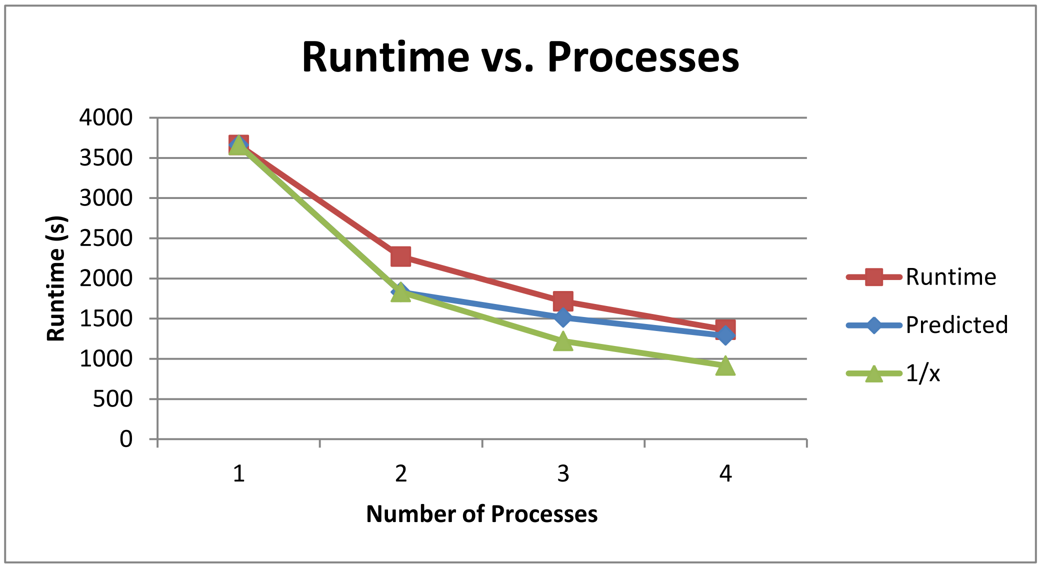


Figure 4.11: Runtime on a Q6600 using insertion mode 2 as the number of processes increases

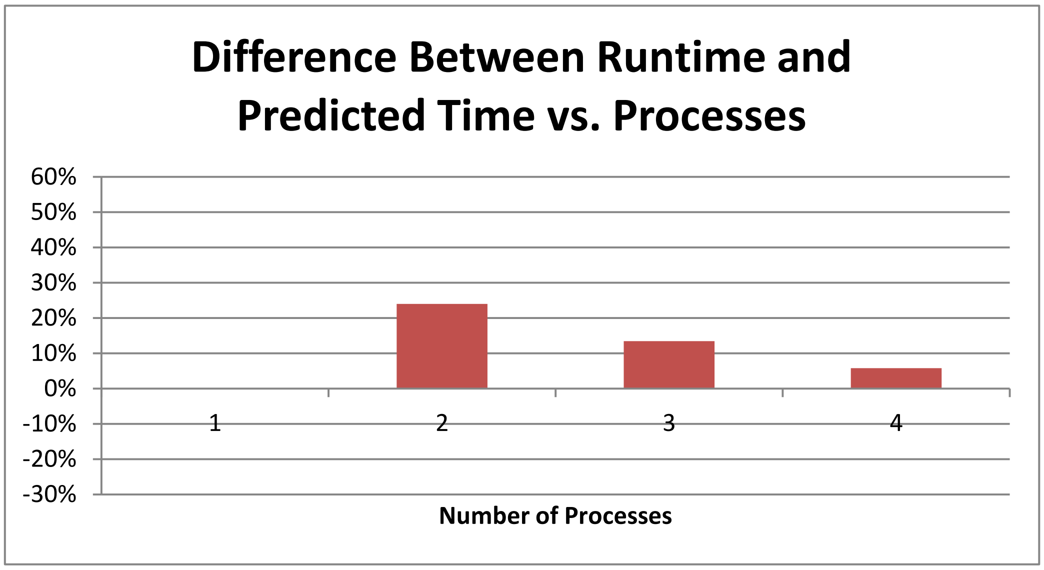


Figure 4.12: Difference between runtime and predicted time on a Q6600 using insertion mode 2 as the number of processes increases

susceptible to scheduling differences, and all of the results from the Q6600, insertion mode 1 has steadily increasing performance compared to insertion mode 2 as the number of processes increases. This increase comes at the cost of increased memory

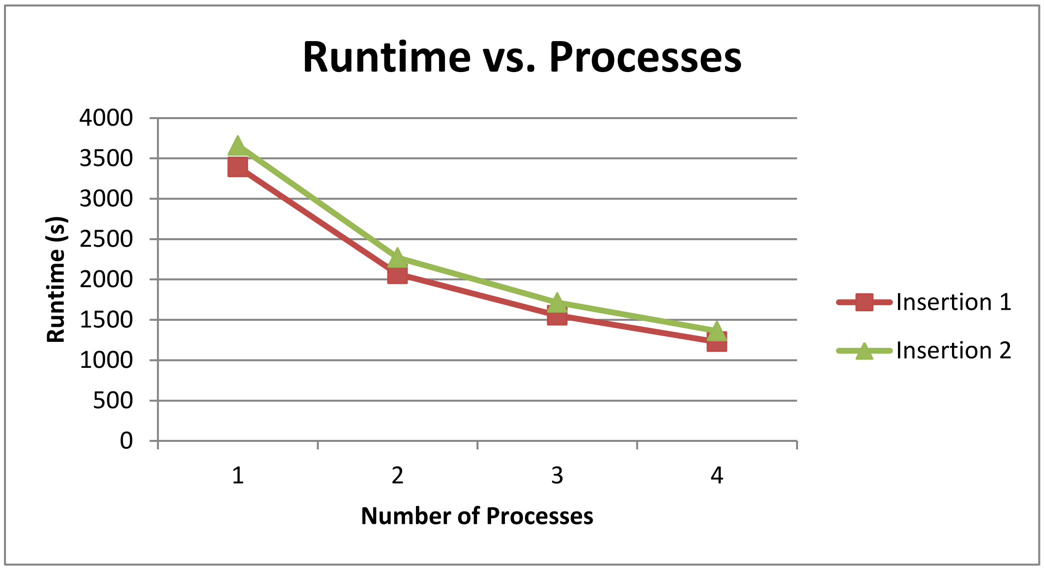


Figure 4.13: Comparison of insertion mode 1 and 2 runtimes on a Q6600 as the number of processes increases

usage. Increasing improvement shows that insertion mode 1 scales better as the number of computers increases than insertion mode 2. The difference in scalability is due to insertion mode 2 needing to perform $2^n - 1$ reads of the input sequence, while insertion mode 1 only needs to perform n reads of the input sequence.

The result of Experiment 4.3.1 is that for both of my insertion modes, with the exception of quirks in the runtime caused by scheduling on Hyper-Threading capable processors, as the number of shared-memory processors used increases, the runtime divides between the available processors. OpenMP, locks, and critical sections cause some overhead, but the majority of this overhead occurs when moving from one to two processes, and only slowly increases when adding additional processes. Equation (4.1) predicts the runtime of a dataset with p processes, r_p, using the runtime of the same dataset running on fewer processes:

$$r_p = \frac{r_{p-1} * (p-1)}{p} \tag{4.1}$$

Applying this equation recursively, until it reaches a known runtime, loses precision with each call due to the precision of the structures used to time the application. Insertion mode 1 performs faster than insertion mode 2, requiring less runtime, but for an input sequence of length l, it requires 2^{l-1} additional bytes of memory per processor, compared to the $2l$ bytes required by insertion mode 2.

For every test with multiple processors, I measured the amount of time each processor spends reading input sequences and inserting their enumerations to the tree. In all tests, each processor required the same amount of runtime. This means that the greedy approach I used for reading rows from the input database, having each processor claim the next row from the database when it finishes its current row, balances the load between processors.

4.3.2 Experiment: Number of Input Sequences

My second experiment measures the effect of increasing the number of rows in the input database on the runtime and memory requirements of my algorithm with both insertion modes. I perform this test using one of the University of Manitoba Bird computers, which have Intel® Core™ i5-661 processors. I use three repeat datasets as input, one with one million, one with ten million, and one with one hundred million rows, with each row containing the same sequence of five items. Since the number of rows in my input datasets increases by a multiple of ten, I predict the runtime of each dataset to be ten times the runtime of the previous. I also include the time to read the input file without mining it to show how this scales with the number of rows.

Figure 4.14 shows the average runtimes and predicted times of my algorithm using insertion mode 1, and Figure 4.15 shows the memory requirements. As the runtime graph shows, the average runtimes of my algorithm match the predicted times, with any differences attributed to rounding and the precision of the timers used to measure runtime. The memory usage remains the same regardless of the number of input rows, since all of the rows contain the same sequence. Similarly, Figures 4.16 and 4.17 show the runtimes and memory requirements of insertion mode 2. The differences between the runtime and predicted times can again be attributed to rounding and the precision of the timers used to measure runtime, and the memory requirements are independent of the number of rows in a repeat dataset.

The time to read the database without inserting is near the time to insert for both insertion modes, again with differences attributed to rounding. This shows that when the number of rows increases by a factor of ten, both the time to read the database and the time to perform insertion, also increase by a factor of ten.

Figure 4.18 compares the average runtimes of insertion modes 1 and 2 as the number of input rows increases. The average runtimes for both insertion modes are the same for the one million and ten million row datasets, but in the dataset with one hundred million rows, the difference between insertion modes becomes visible. As in the experiments with the number of shared memory processors, insertion mode 1 requires less runtime than insertion mode 2, at the cost of increased memory as seen in Figure 4.19.

The result of Experiment 4.3.2 is that the runtime of both of my insertion algo-

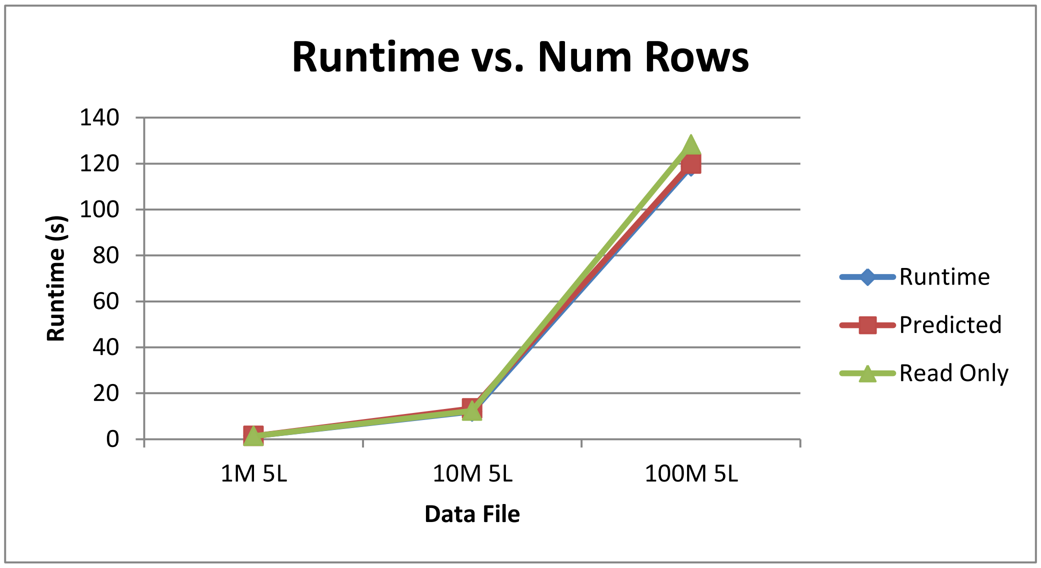


Figure 4.14: Runtime of insertion mode 1 as the number of input rows increases

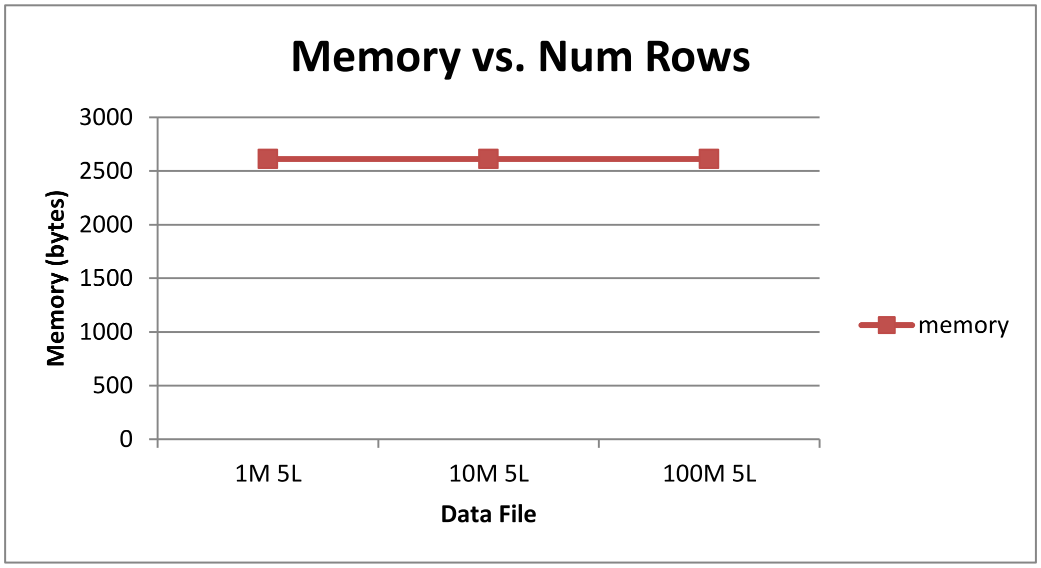


Figure 4.15: Memory use of insertion mode 1 as the number of input rows increases

rithms is the time it takes them to insert one row, multiplied by the number of rows in the input dataset. Equation (4.2) predicts the runtime with w rows, r_w, based on runtimes of datasets with the same number of items per row and fewer rows:

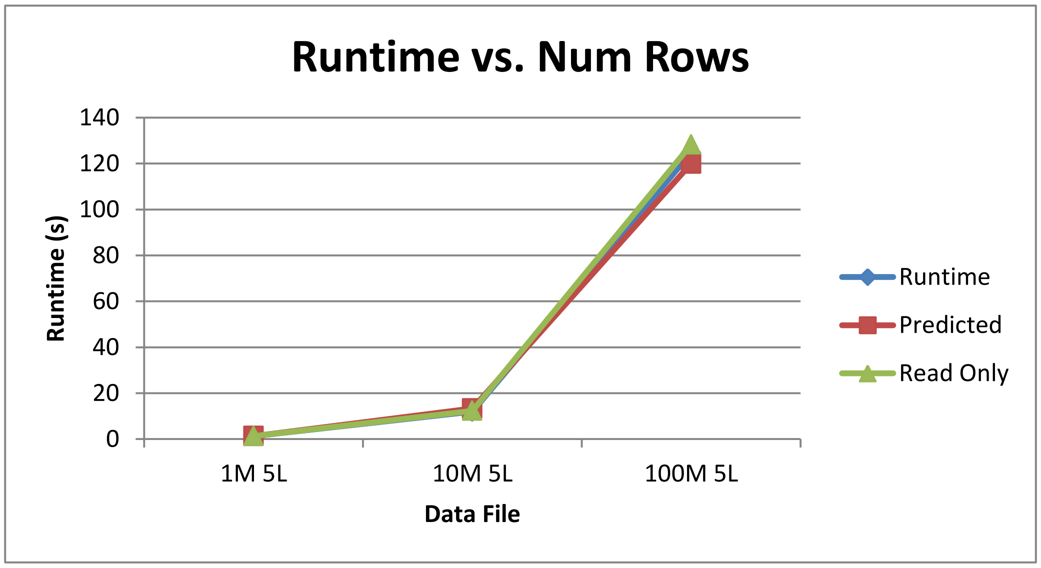


Figure 4.16: Runtime of insertion mode 2 as the number of input rows increases

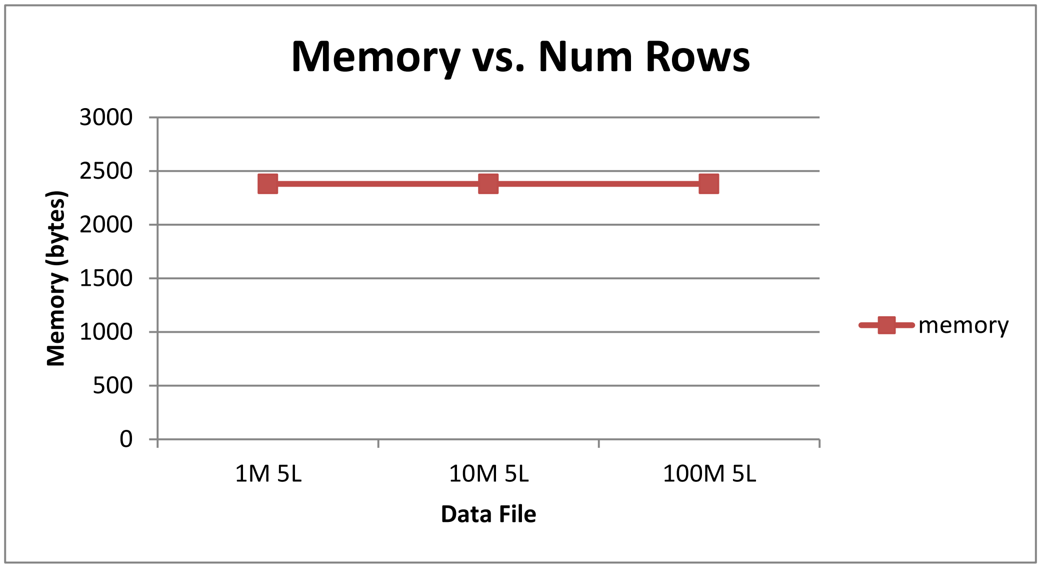


Figure 4.17: Memory use of insertion mode 2 as the number of input rows increases

$$r_w = \frac{r_{w-1} * w}{w - 1} \tag{4.2}$$

This same result holds for the read only test. The memory requirements of my

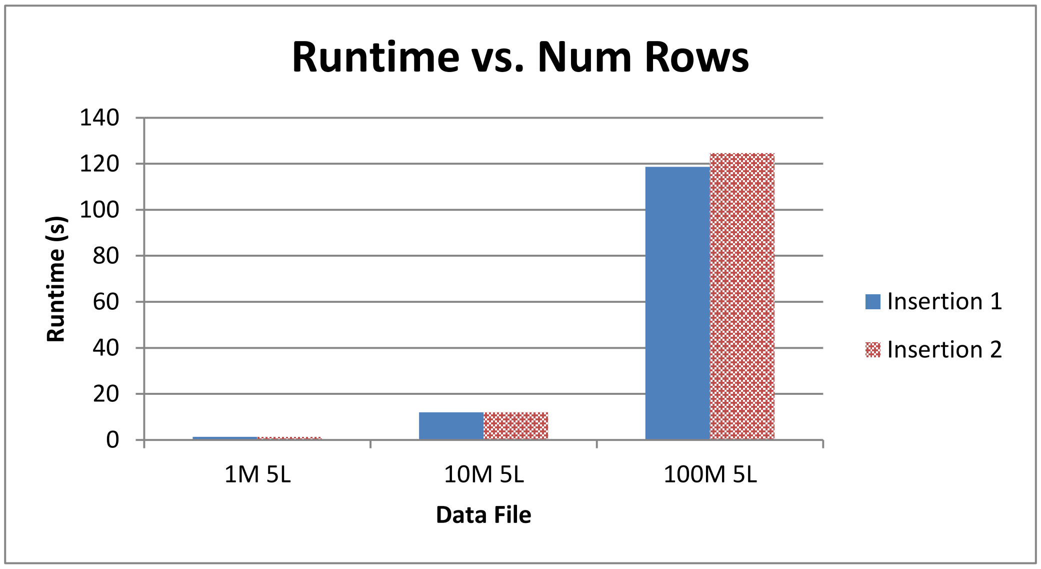


Figure 4.18: Comparison of the runtimes of insertion modes 1 and 2 as the number of input rows increases

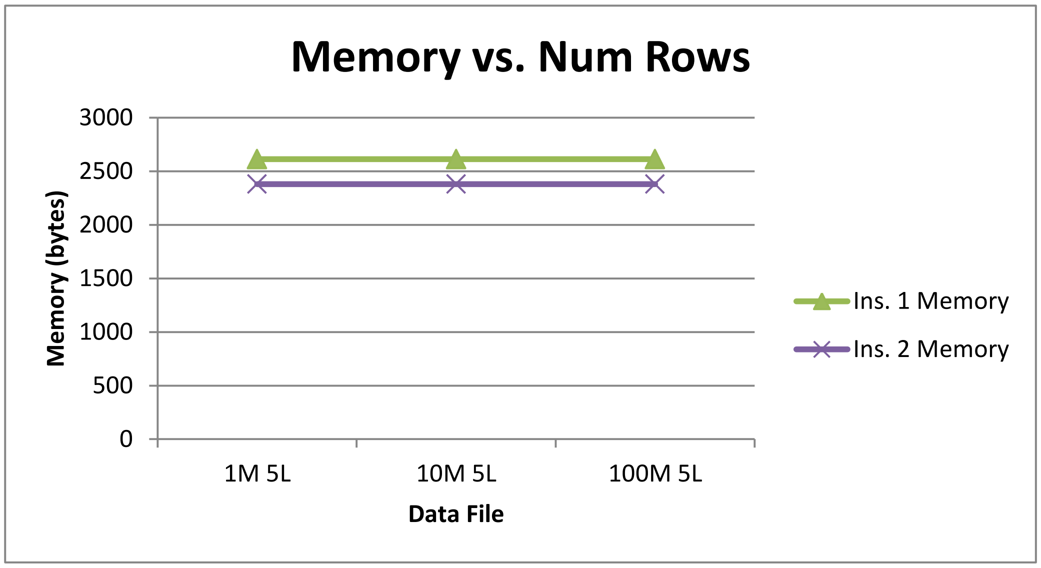


Figure 4.19: Comparison of the memory requirements of insertion modes 1 and 2 as the number of input rows increases

insertion modes do not change as the number of input rows increases. Insertion mode 1 performs better than insertion mode 2, requiring less runtime, but it requires more memory to achieve this.

4.3.3 Experiment: Input Sequence Length

My third experiment measures the effect changing the number of items in each input sequence has on the runtime of my algorithm. I test both of my insertion modes for this using one of the University of Manitoba bird computers. I use six repeat datasets, each with ten thousand transactions, and identical sequences ranging from fifteen to twenty items in length. Since both of my insertion algorithms require $2^l - 1$ node reads and writes, where the number of items in the input sequence is l, I predict the runtime will double with each increment of the input sequence length. I also measure the time it takes to read the input file without mining it.

Figure 4.20 and Figure 4.21 show the average runtimes of my algorithm using insertion modes 1 and 2 respectively. For both modes, the average runtime matches the predicted time, with any differences attributed to rounding and precision of the timers used to measure runtime. The memory usage, both the number of memory allocations and the number of bytes used, also doubles with each increase in sequence length. The time to read the input dataset grew much more slowly, taking an average of 3 seconds for the length 15 and 16 sequences, 3.3 seconds for length 17 and 18 sequences, and 3.6 seconds for length 19 and 20 sequences. Figure 4.22 and Figure 4.23 show the memory used by the first and second insertion modes.

Figure 4.24 compares the average runtimes, and Figure 4.25 compares the memory requirements, of insertion modes 1 and 2. As the length of the input sequence increases, the distance between the average runtimes also increases, with insertion mode

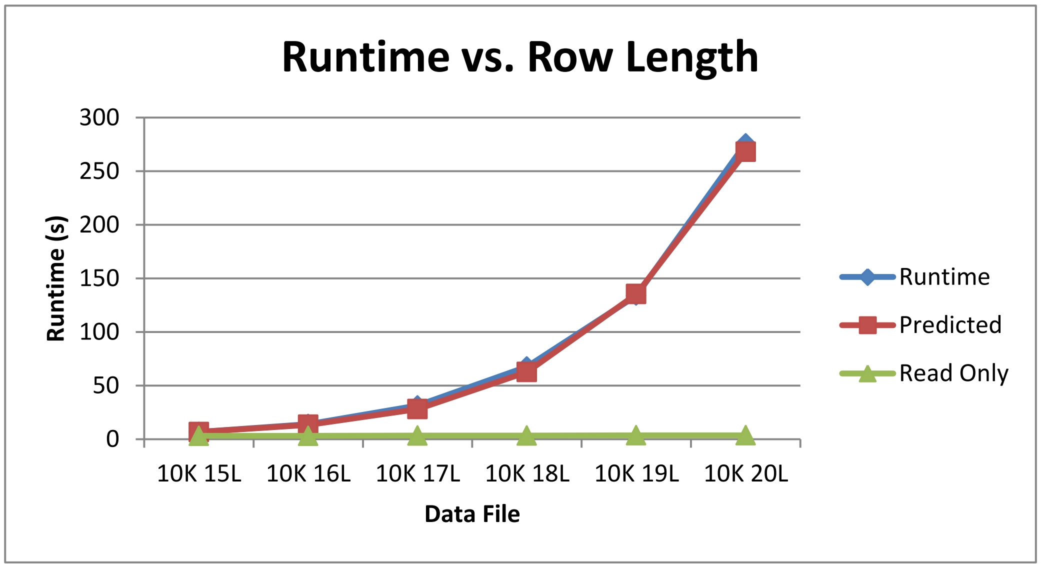


Figure 4.20: Runtime of insertion mode 1 as the input row length increases

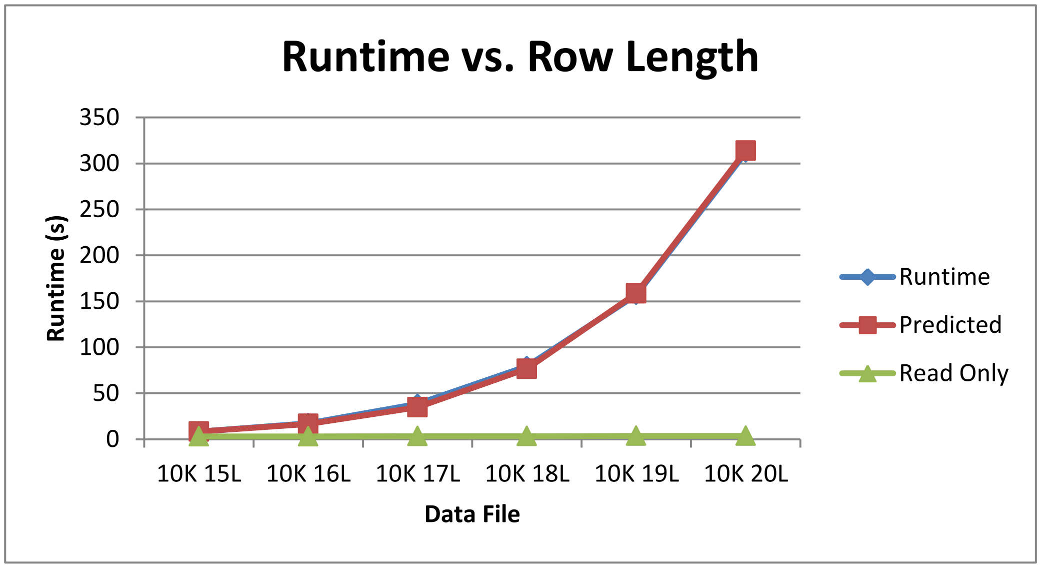


Figure 4.21: Runtime of insertion mode 2 as the input row length increases

1 consistently requiring less runtime than mode 2. Insertion mode 1 requires 2 seconds less runtime than insertion mode 2 with an input sequence of length 15, and this distance increases to 19 seconds less for an input sequence of length 20. This increase in performance comes at the cost of increased memory usage. The enumeration-trees

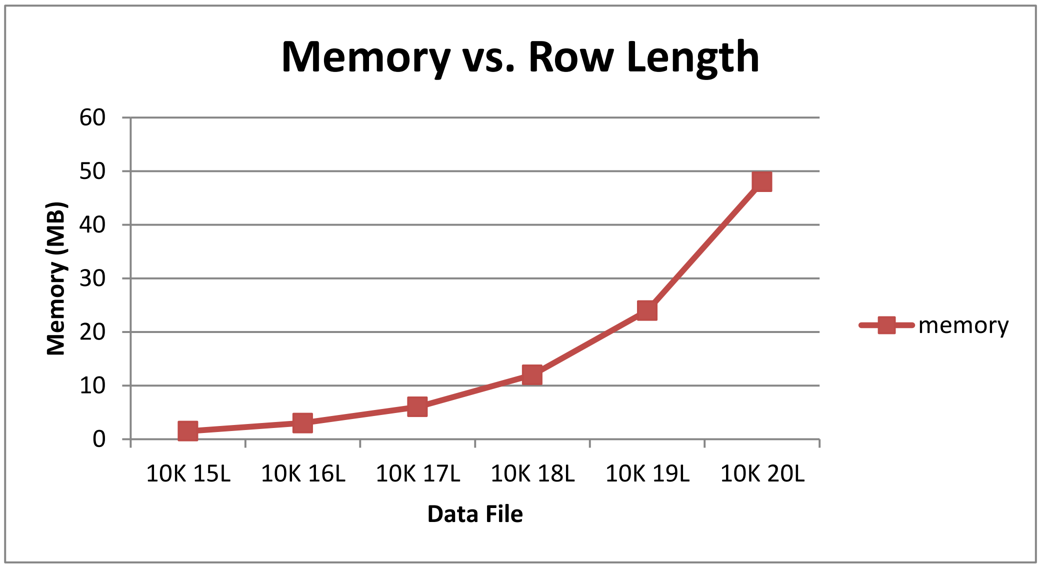


Figure 4.22: Memory use of insertion mode 1 as the input row length increases

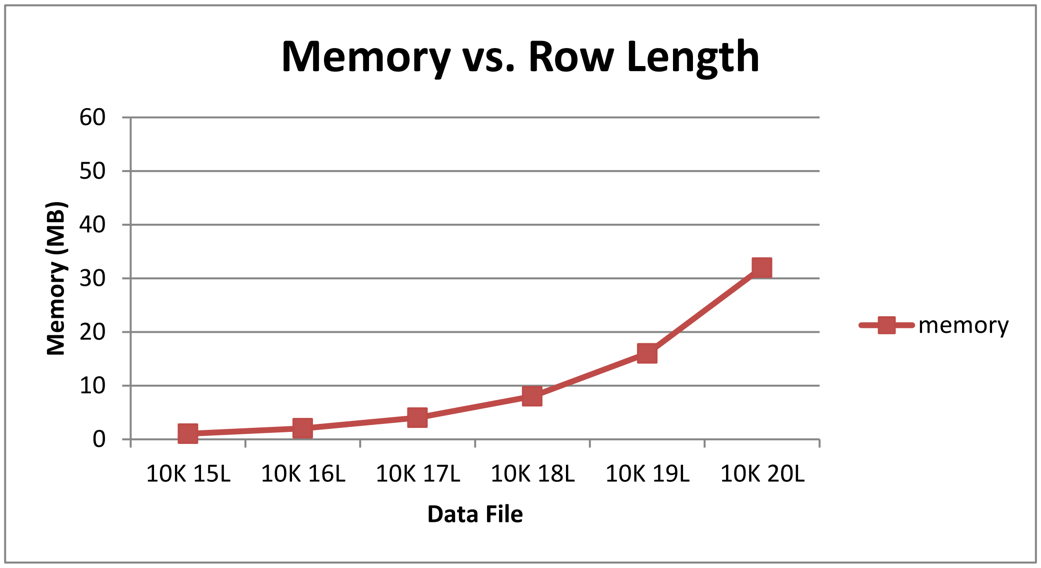


Figure 4.23: Memory use of insertion mode 2 as the input row length increases

increase in size equally for both insertion modes, but insertion mode 1 doubles the size of its insertion list, while insertion mode 2 increases the size of its stack by one.

The result of Experiment 4.3.3 is that the runtime of both of my insertion algorithms follows their predicted values, doubling every time the number of items in the

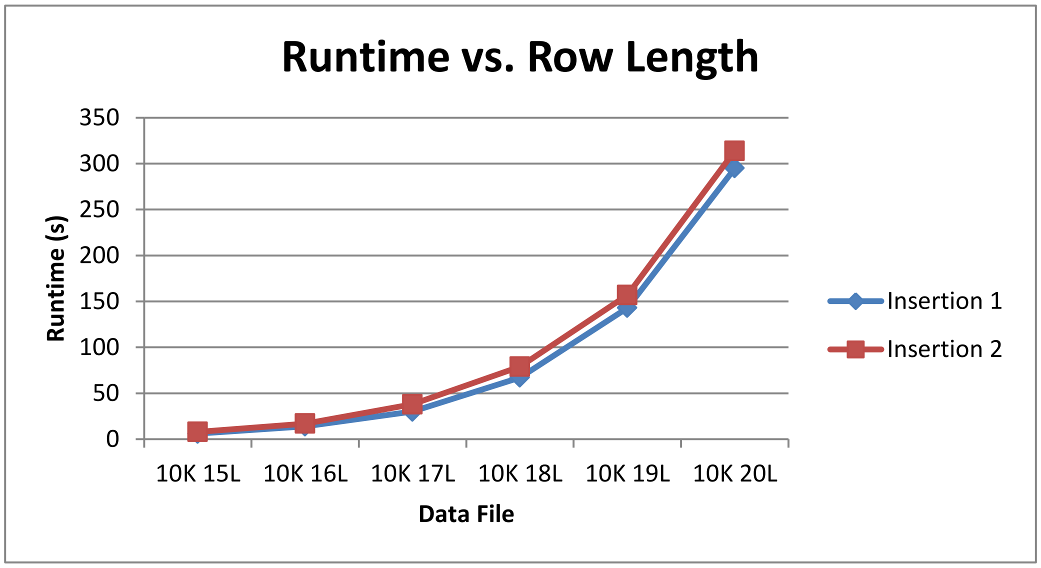


Figure 4.24: Comparison of runtimes of insertion modes 1 and 2 as the input row length increases

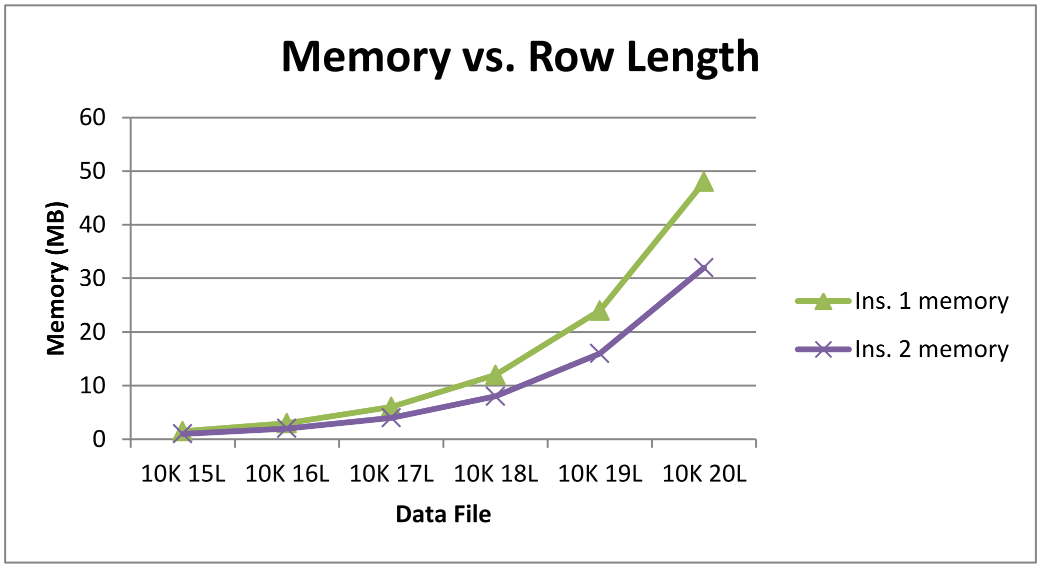


Figure 4.25: Comparison of the memory requirements of insertion modes 1 and 2 as the input row length increases

input sequences increases, so the runtimes are the time to insert one item multiplied by 2^l, where l is the number of items in the input sequence. Since my algorithm performs 2^l insertions for each row, and doubles the runtime as a result, this suggests that algorithms that transform the database prior to mining may be more suited to mining long sequences. Transforming the database overlaps identical prefixes of the input sequences, effectively reducing the number of unique rows in the database to mine.

Equation (4.3) predicts the runtime of a dataset with sequences of length l given the runtime of a dataset with the same number of rows and sequences of $l-1$ items:

$$r_l = r_{l-1} * 2 \tag{4.3}$$

Insertion mode 1 performs faster than insertion mode 2, since it only performs l reads of each input sequence, or one read per item, compared to the 2^l reads insertion mode 2 performs, but requires 33% more memory. While the insertion time increases exponentially, the read only time increases linearly, as it does when the number of rows increases.

4.3.4 Experiment: Result Collection

Experiment 4.3.4 compares the performance of my two collection modes. Collection mode 1 collects only potentially frequent sequences, and collection mode 2 collects all sequences. I perform this experiment on my four heterogeneous computers using the synthetic dataset, since it is the only dataset affected by minimum support.

Collection mode 1 works by sending any locally frequent sequence to the collecting computer, as soon as it becomes frequent, so that the collecting computer can flag it as potentially being globally frequent. Once all computers have completed sub-sequence enumeration and inserted the corresponding nodes into the enumeration tree, the collecting computer performs a depth-first transversal of only potentially frequent nodes. Each time the search visits a node, the collecting computer sends a request for counts of that node to each distributed computer. The transversal moves to a node's potentially frequent children only if it is frequent after collection.

This collection method has three different components. Two, tracing and sending locally frequent sequences, take place during the insertion process. The third component, collecting potentially frequent results, takes place after the insertion process. Figure 4.26 shows the runtime of the components of collection mode 1 as the minimum support increases. The time spent inserting sub-sequences to the enumeration tree remains constant as the minimum support decreases, since my algorithm always enumerates all sub-sequences. The time spent tracing and sending newly frequent sub-sequences to the collecting computer is under one second for all tests. The time spent collecting results is under one second until the minimum support threshold reaches 10%. As the minimum support threshold decreases from 10% to 1%, the time spent collecting results increases. This increase follows the number of nodes collected, shown in Figure 4.27. This cost comes from the need for both a broadcast and reduction operation for each node collected.

Collection mode 2 works by serializing the local result trees and saving them to disk, where the collecting computer reads them as it has a processor available to do so. This collection mode has two components, the time each computer spends saving serialized trees to disk, and the time the collecting computer spends loading serialized trees. Since minimum support does not reduce the size of the serialized trees, it does

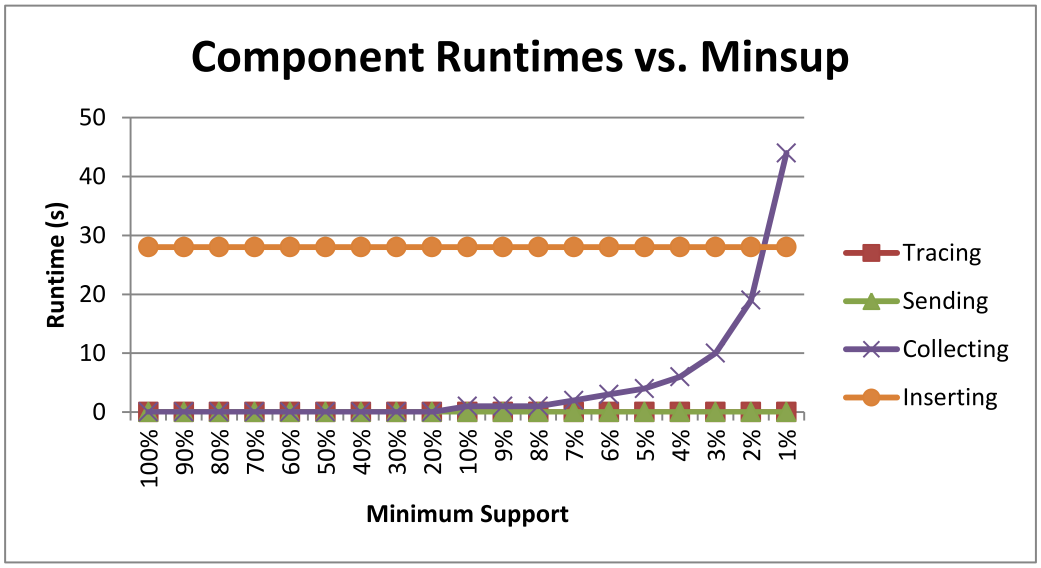


Figure 4.26: Runtime of collection mode 1 as the minimum support decreases

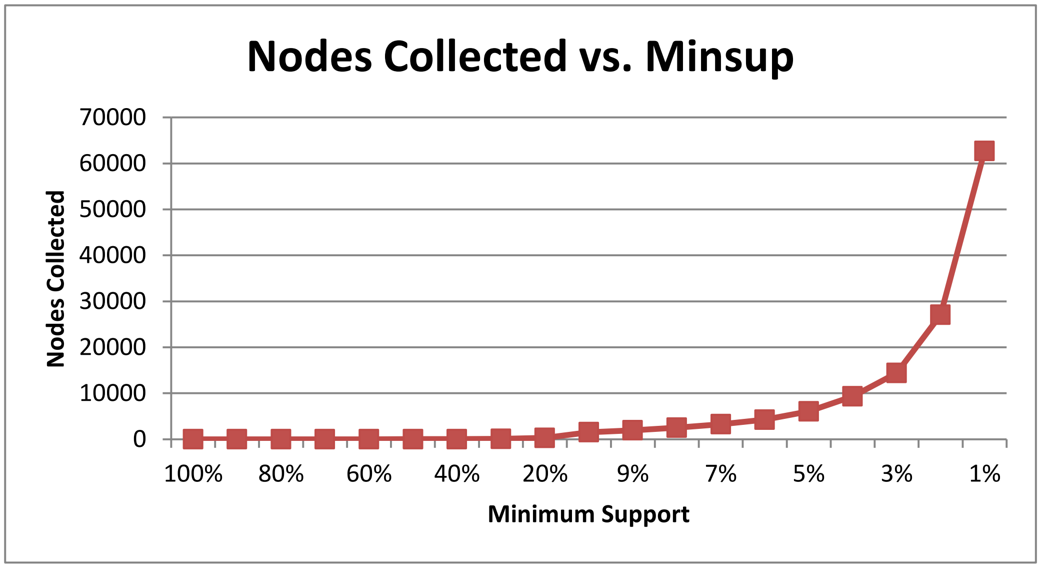


Figure 4.27: Number of nodes collection mode 1 collects as the minimum support decreases

not have any effect on the resulting runtimes. Figure 4.28 shows this.

Figure 4.29 compares the total runtimes of collection modes 1 and 2. While the minimum support value is above 10% and the number of frequent tree nodes is low,

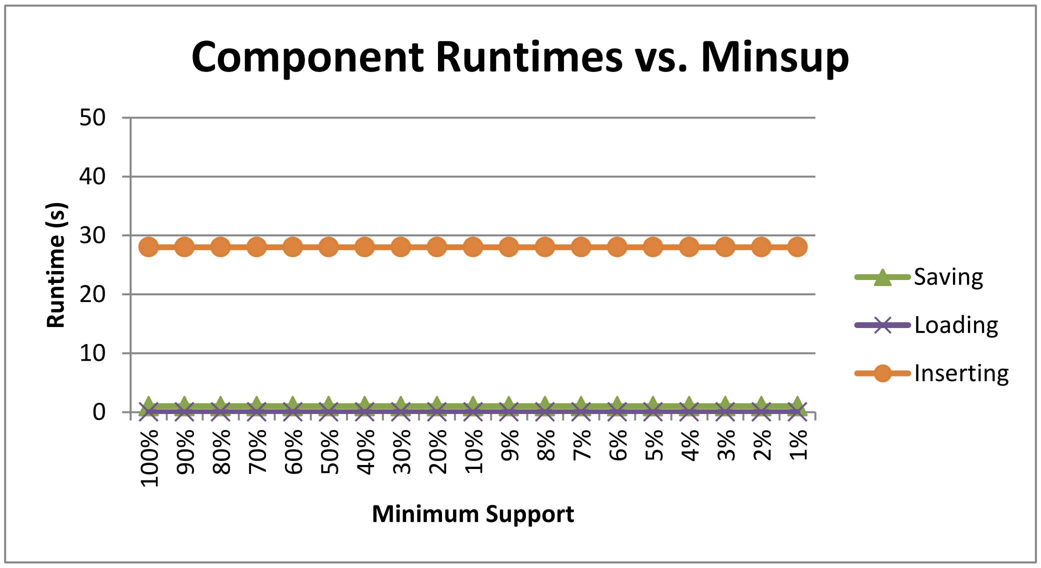


Figure 4.28: Component runtimes of insertion mode 2 as the minimum support decreases

collection mode 1 performs faster than collection mode 2. Both collection modes have the same performance with minimum supports from 10% to 8%. As the minimum support drops below 8%, and the number of frequent tree nodes is high, the runtime collection mode 1 requires is higher than collection mode 2.

The result of Experiment 4.3.4 is that collection mode 1 has better performance than collection mode 2 as long as the minimum support value is high. If a user needs to mine with a low minimum support threshold, they should use collection mode 2, as collection mode 1 has poor performance in this case. For my synthetic dataset, a minimum support value of 8% or higher requires collection mode 1 to collect at most 2504 nodes, and the runtime is less than or equal to that of collection mode 2. As the minimum support threshold drops below 8%, collection mode 1 takes more runtime than collection mode 2, due to the number of MPI broadcast and reduction operations it performs. The minimum support value does not affect the runtime for

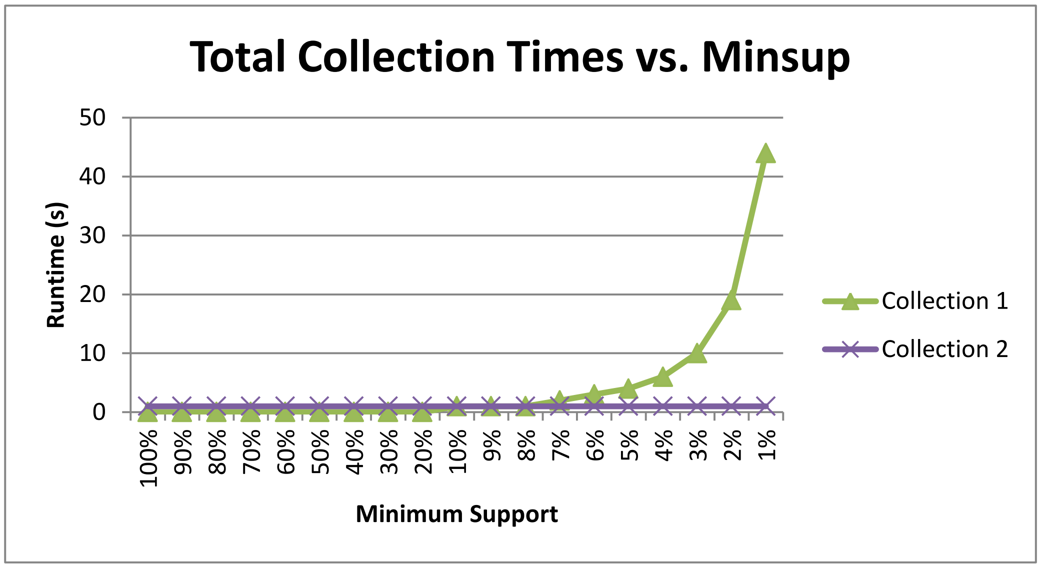


Figure 4.29: Comparison of collection mode 1 and 2 runtimes as the minimum support decreases

collection mode 2, as it always collects the entire result set from each computer. Even though collection mode 2 has more data to collect, it can perform this collection using a file stream, which only has to initiate and finalize communication once, rather than a series of separate communications through MPI.

4.3.5 Experiment: Load Balancing

My fourth experiment compares the runtimes on my four heterogeneous computers to show the effectiveness of my load balancing technique. I use three input datasets for this experiment. Two are repeat datasets with ten thousand rows and sequence lengths of 15 and 20 items, and the third is the synthetic dataset.

As I presented in Section 3.3, my load balancing mechanism works by running the same dataset on each computer to determine the relative performance of each computer. Using this method, I do not partition the input datasets between the computers for this experiment, but rather assign the entire dataset to each computer. I measure how long it takes each computer to enumerate all sub-sequence from the input dataset. My load balancing algorithm uses these runtimes to determine the portion of input data that it should assign to each computer. I measure the overall accuracy of my load balancing mechanism by comparing the portion sizes it creates for each input dataset.

Figure 4.30 shows the average runtimes of the three input datasets on each of my four Intel®-based heterogeneous computers. For all datasets, the Core^{TM} i7-920 requires the least runtime, followed by the Core^{TM} i3-530, the Core^{TM} 2 Q6600, and the Core^{TM} 2 E5200. Figure 4.31 shows how much the average runtimes change between datasets. Comparing the repeat datasets with 15- and 20-item sequences, the i7-920, i3-530, and E5200 have the same change in average runtime, with their 15-item sequence datasets requiring 3% of the time their 20-item sequence datasets do. The Q6600 changes slightly more, with its 15-item sequence dataset requiring 4% of the time its 20-item sequence dataset does.

There is more variation when comparing the repeat set with 15-item sequences to the synthetic dataset. While both the i7-920 and the E5200 have average runtimes for their 15-item sequence datasets that are 7% of their synthetic average runtimes, the ratio on the i3-530 is 5%, and the ratio on the Q6600 is 9%. Similarly, comparing the repeat set with 20-item sequences with the synthetic dataset, the i7-920, Q6600, and E5200 have similar ratios of 39%, 44%, and 41%, but the i3-530 has a ratio of 53%. The runtime changes between datasets remains level across the four computers, with the exception of the change between the synthetic dataset and repeat dataset with

20-item sequences on the i3-530. With this exception, the largest difference between change ratios is 5, between the i7-920 and the Q6600 when comparing the synthetic dataset to the repeat dataset with 20-item sequences.

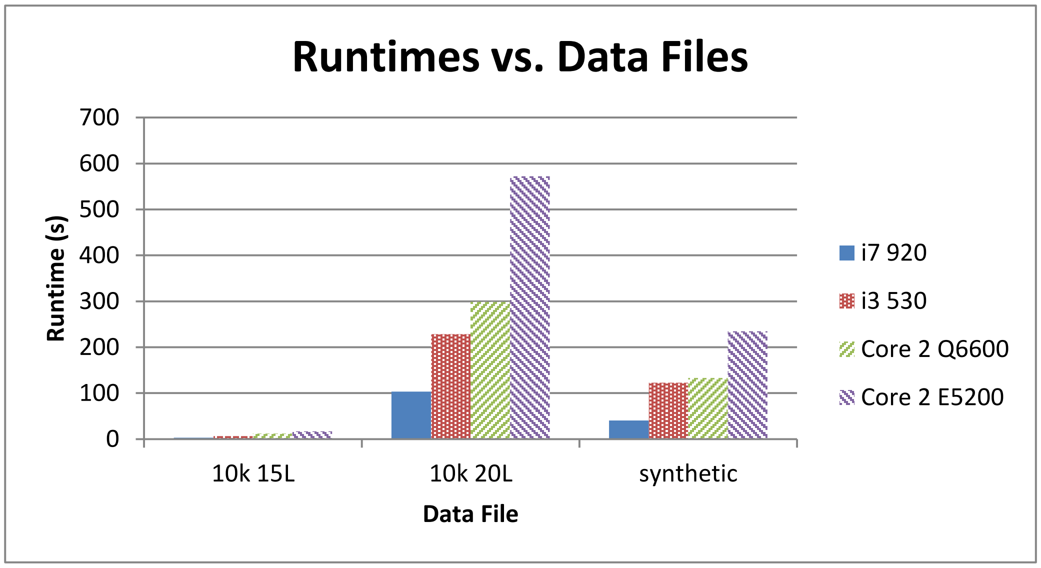


Figure 4.30: Runtime comparison between heterogeneous computers

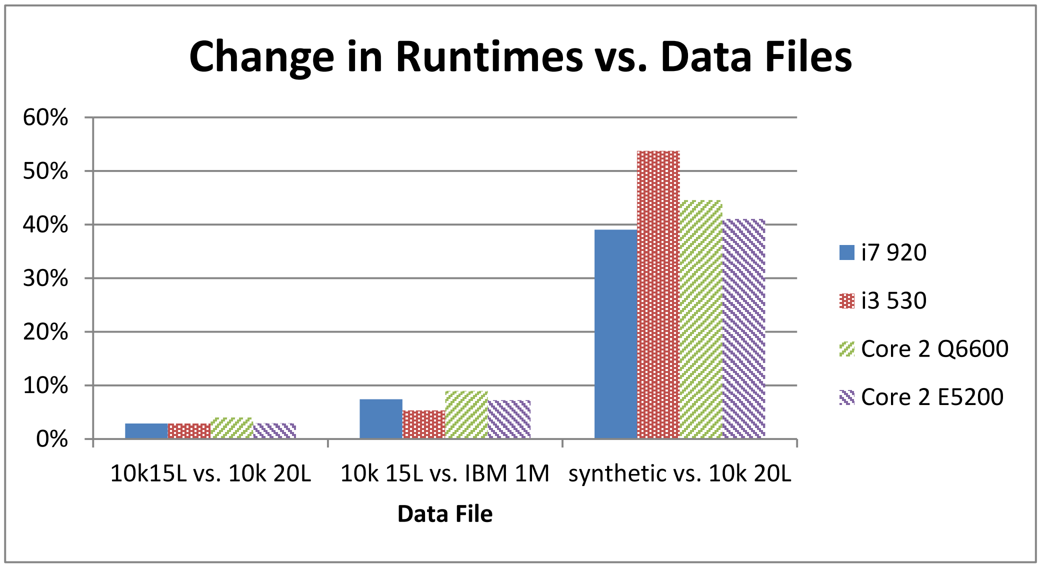


Figure 4.31: The percentage change of runtimes between input datasets

Figure 4.32 shows the portion sizes that my load balancing algorithm assigns based on the repeat datasets and the synthetic dataset. The portion sizes do not stay constant for all three datasets, but the average difference between portion sizes is 2.5, so the portion sizes assigned from each dataset remain close. My algorithm saves these portions to use for future runs so that it does not have to perform load balancing during the mining process. As I have presented in Section 2.2, algorithms that provide a reasonably balanced load without impacting performance have better overall performance than algorithms that balance the load perfectly at high cost.

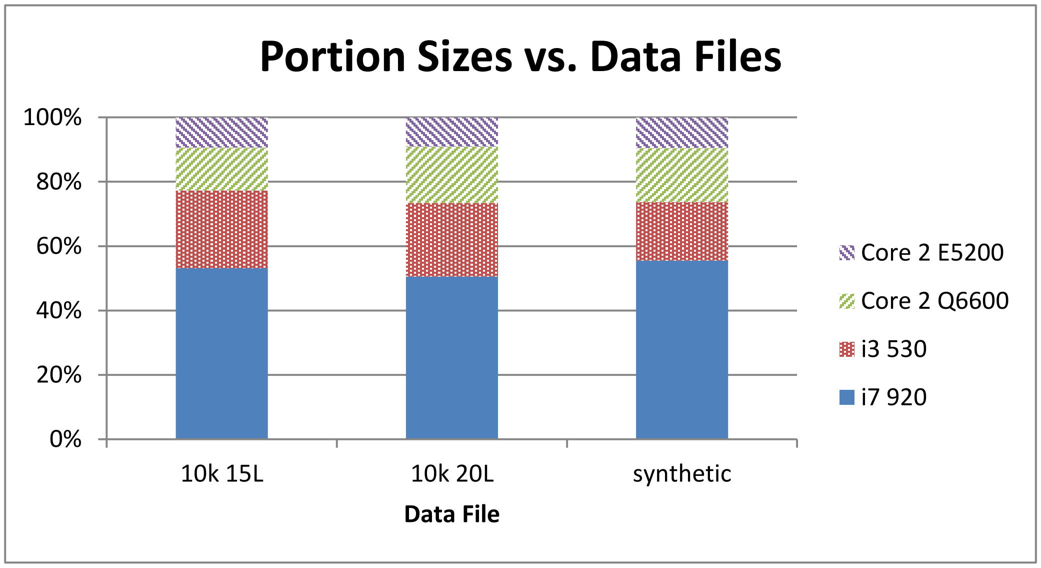


Figure 4.32: Portion sizes based on three input datasets

The result of Experiment 4.3.5 is that while my load balancing mechanism does not provide a perfectly balanced load, the portion sizes remain close at an average distance of 2.5 from each other, which will provide a near-balanced load. As my load balancing algorithm only needs to load the portion sizes from a previous benchmark and multiply them by the total number of rows when mining, it requires essentially no overhead and will not impact scalability. As I found in Section 2.2, an algorithm with a near-balanced load and low load balancing overhead typically requires less total runtime than an algorithm with a perfectly-balanced load and high load balancing overhead.

4.3.6 Experiment: Number of Computers

Experiment 4.3.6 measures the effect of increasing the number of computers used on the runtimes of my algorithm. I perform this experiment using 1 to 24 University of Manitoba bird computers. I use the repeat dataset with ten thousand rows of 20 items. This puts an emphasis on row length rather than the number of rows. I emphasize this because the number of items collected has an effect on the collection methods, where the number of rows does not. I measure the runtimes of both collection modes to show how they scale with the number of computers. I use insertion mode 1 for both tests as it consistently outperforms my second insertion mode. The amount of memory used remains constant, as the number of computers does not affect it. I also measure the time to read the database without mining.

Figures 4.33 and 4.34 show the average component runtimes of collection mode 1 as the number of computers increases. Since increasing the number of computers decreases the number of rows assigned to each computer, the time spent inserting sequences matches the predicted time, as it does in Experiment 4.3.2. The time to read the database also scales as expected, as it does when changing the number of rows in Experiment 4.3.2. The time my algorithm spends flagging frequent sequences on the collecting computer stays low until the number of computers reaches 22, where

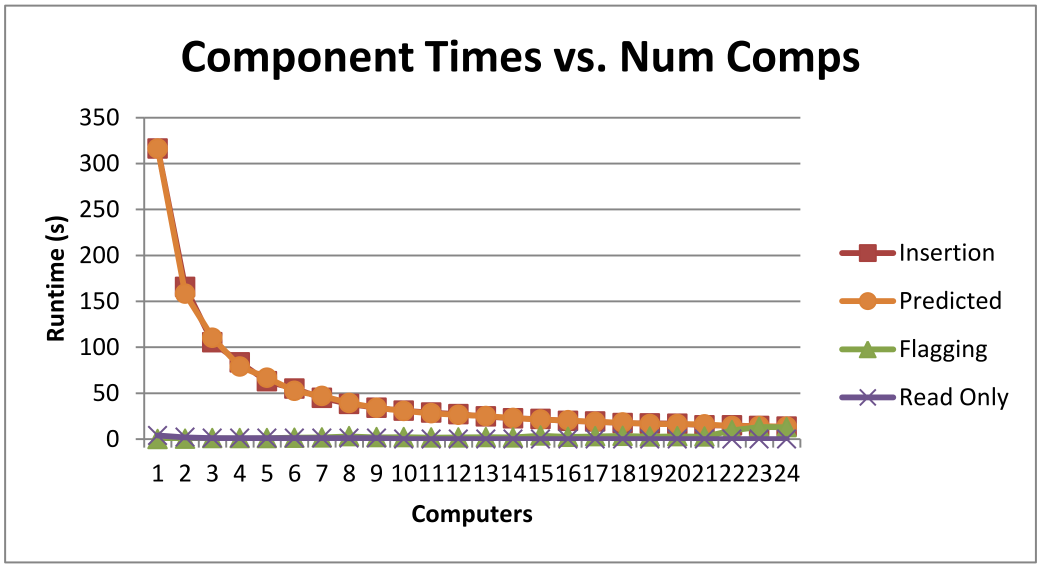


Figure 4.33: Component runtimes of collection mode 1 as the number of computers increases

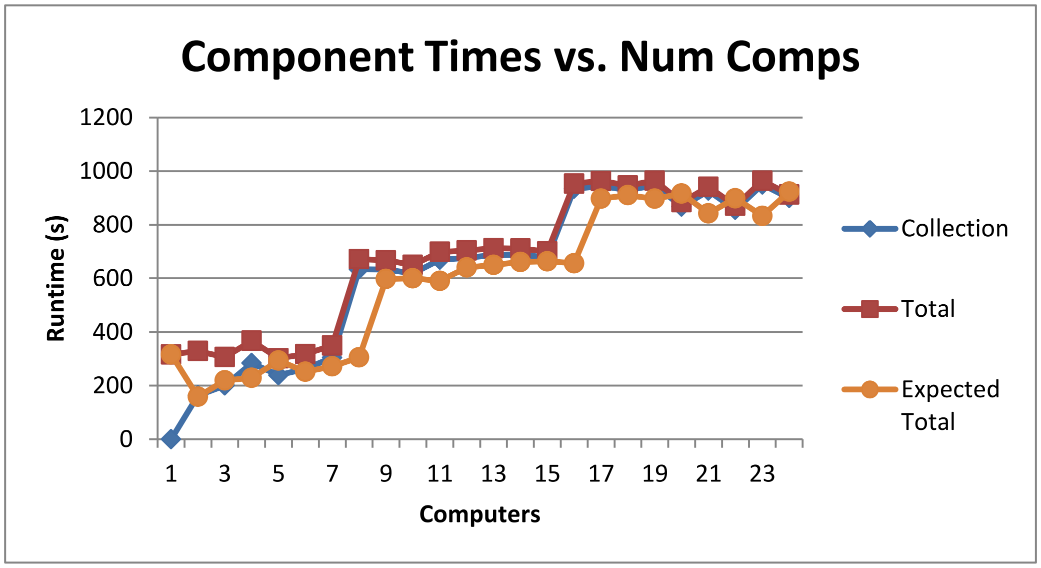


Figure 4.34: Component runtimes of collection mode 1 as the number of computers increases

it jumps from 2 to 9 seconds. This jump indicates that the flagging computers have saturated the receiving process on the collecting computer, and need to wait for it to become available.

I expected the runtime of c computers, r_c, to be $r_c = r_{c-1} * (c-1)/c$, but Figure 4.34 shows that this is not the case. The total runtime is the sum of the insertion, flagging, and collection times. As the number of computers increases, the cost of the broadcast and reduction operations in the results collection stage also increases. Rather than decreasing as the number of computers increases, the total runtime increases as a series of three steps. Note that this test is using a repeat dataset, so my first collection method has to collect all nodes. I found that my first result collection method does have good performance when few nodes are frequent in Experiment 4.3.4.

On the first step, using 1 to 7 computers, the total runtime remains level, since the increasing communication balances out the decreasing insertion time. I would expect this trend to continue, with the total runtime remaining the same, but Figure 4.34 again shows that this is not the case. When I add the 8^{th} computer, the average collection time jumps from 304 seconds to 633 seconds, and when I add the 16^{th} computer, the average collection time jumps from 678 seconds to 953 seconds. These two jumps are the start of the second and third level steps in the collection time graph. Since the reduction that collects the global counts of each potentially frequent node requires every computer to finish sending before it completes, these steps show that computers 8 and 16 are slower than the others, and are limiting performance.

Figure 4.35 shows the total average runtime of collection mode 1 as the number of computers increases, and shows how much of the runtime comes from each component. The time each computer spends inserting nodes decreases as the number of computers increases. The collection time, on the other hand, generally increases as the number of

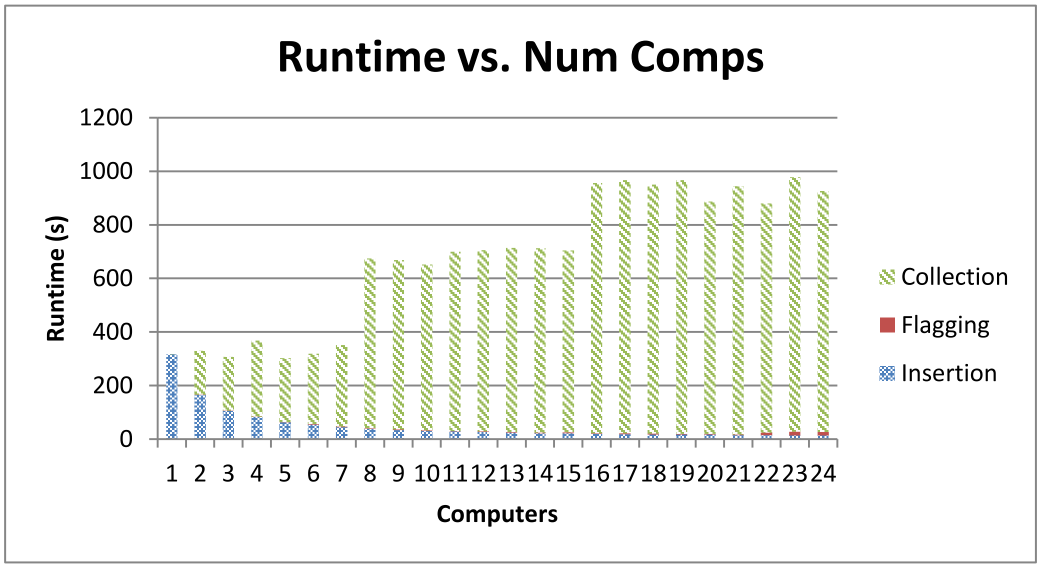


Figure 4.35: The total runtime of collection mode 1 as the number of computers increases

computers increases. The time spent flagging frequent sequences during the insertion remains low using 1 to 21 computers, with an average flagging time of 2 seconds. The flagging time jumps from 2.7 seconds with 21 computers to 9.6 seconds with 22 computers, and takes an average of 11.9 seconds between 22 and 24 computers. This increased flagging time is small, relative to the average collection time between 16 and 24 computers, which is 917 seconds.

Figure 4.36 shows the component runtimes of collection mode 2 as the number of computers increases. The average insertion time follows its predicted time, as increasing the number of computers reduces the number of input rows, as in Experiment 4.3.2. Unlike collection mode 1, the average total runtime follows the expected total runtime, decreasing with each additional computer. The total runtime of the collecting computer is its insertion time followed by the time it spends waiting for, reading, and inserting serial files. The total runtime of the other computers is the

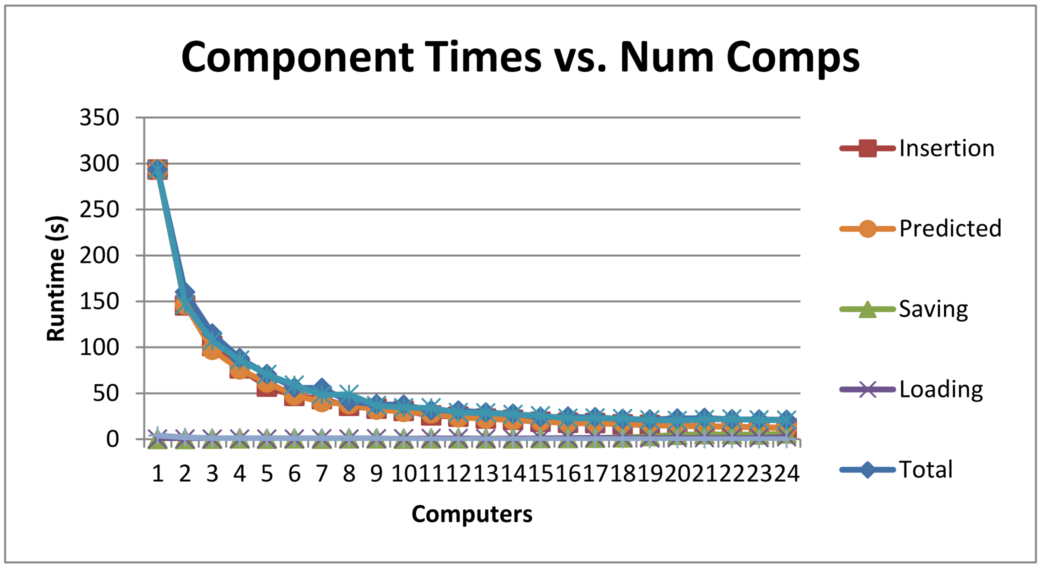


Figure 4.36: Component runtimes of collection mode 2 as the number of computers increases

time they spend inserting sequences, followed by the time to serialize their results to disk. The time spent saving and loading files remains near zero throughout the range, which makes insertion take up the majority of the time in my algorithm.

All computers complete mining and start serializing their trees within a second of each other, since dividing a repeat set evenly over homogeneous computers gives a perfectly balanced load, making this the worst case for the saving times. The saving and loading times remain constant until 14 computers are used, at which point the computer storing the serialized results cannot keep up with the incoming data, and the average time each computer spends saving its serial file increases. Figure 4.37 shows the detailed saving and loading times as the number of computers increases.

Some load imbalance and having the collecting computer complete mining first would be ideal for collection mode 2. It would avoid having all computers save at the same time, and the collecting computer would be able to start loading serial files

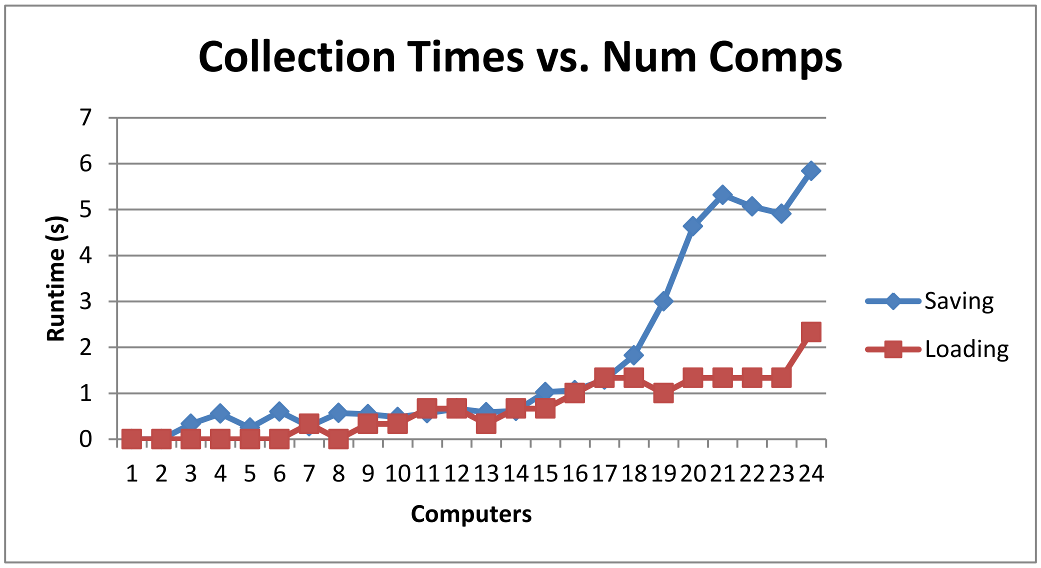


Figure 4.37: Detailed view of the saving and loading times required by collection mode 2 as the number of computers increases

while it waits for other computers to complete. Even though collection mode 2 writes serialized files to disk, rather than copying trees straight from memory to memory in collection mode 1, writing to a file stream is much more efficient than multiple MPI broadcast and reduction operations.

Figure 4.38 compares the total average runtimes of collection mode 1, collection mode 2, and the read only time, as the number of computers increases. While the communication cost cripples the performance of collection mode 1, the runtime of collection mode 2 approaches the read only time as the number of computers increases.

The result of Experiment 4.3.6 is that the overhead of the MPI broadcast and reduction operations collection mode 1 uses negates the impact of reducing the number of nodes it has to collect. This communication causes the runtime of collection mode 1 to increase as the number of computers increases when there are many nodes to collect. As in the collection mode tests, reducing the number of potentially frequent

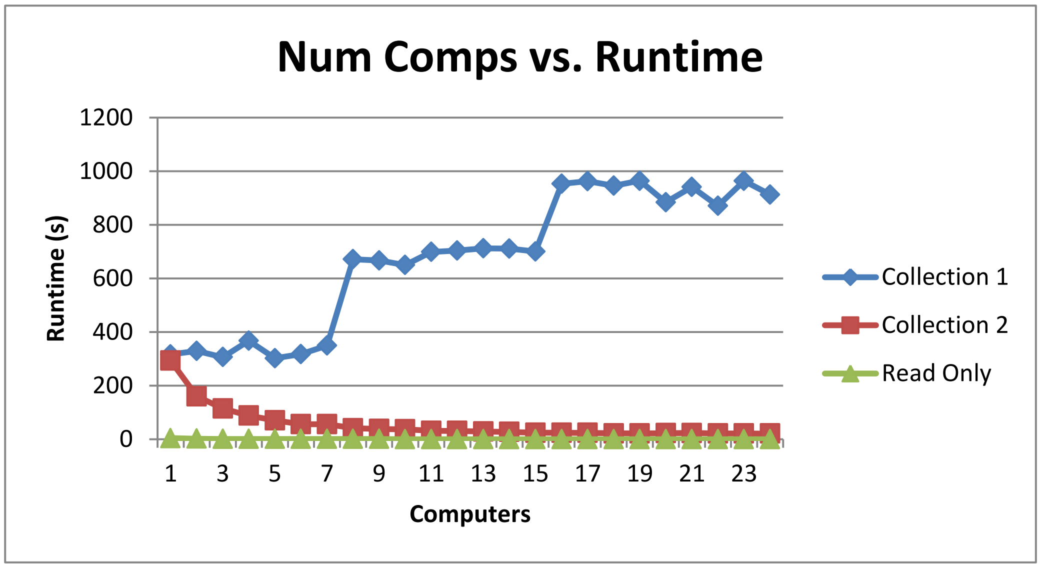


Figure 4.38: Comparison of the average runtimes of collection modes 1 and 2 as the number of computers increases

nodes would reduce this communication cost. Collection mode 2 performs much better as the number of computers increases, with the sub-sequence enumeration time scaling as expected, and the communication cost remaining low. Equation (4.1), the function that predicts the performance as the number of processors increases, works for collection mode 2 on homogeneous computers. When all computers are the same, my load balancing algorithm partitions the input data evenly, so the total runtime splits evenly between computers. On heterogeneous computers, where my load balancing algorithm assigns portions with different sizes based on the computers performance, the equation used to predict the runtime based on the number of rows predicts the performance of collection mode 2, when applied to any computer and its balanced dataset.

4.3.7 Experiment: Comparison with PrefixSpan

In my final experiment, I compare the performance of the serial PrefixSpan algorithm with my algorithm using insertion mode 1, my insertion list, and collection mode 2, my serialization method. I perform these experiments on my cluster of four Microsoft® Windows® computers. I perform the single computer tests on my Core^{TM} i7 computer, and use my Core^{TM} i3 computer as the central database.

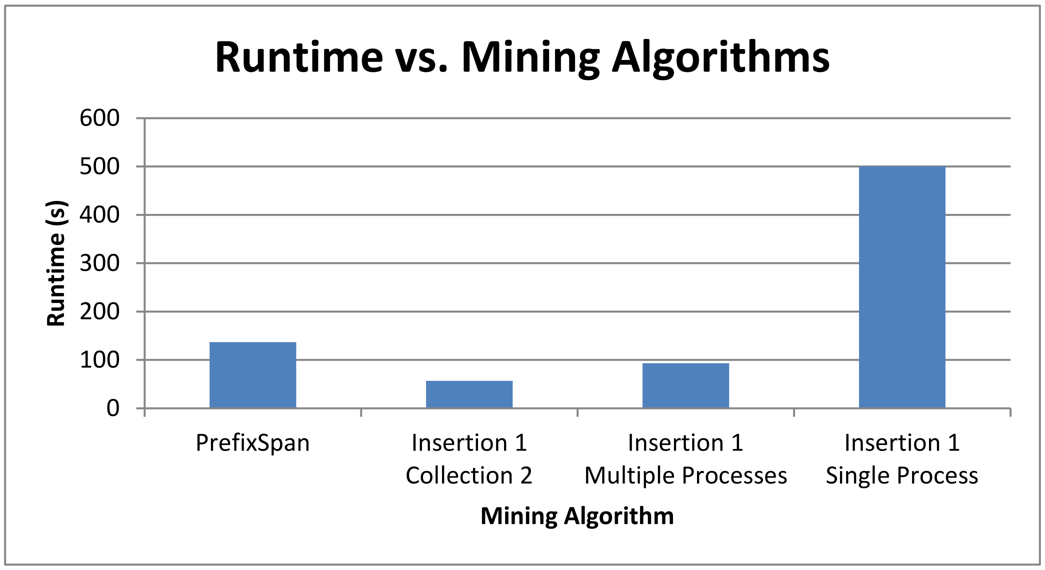


Figure 4.39: The runtime of PrefixSpan in comparison to the runtime of my algorithm using four heterogeneous computers, one computer with multiple processes, and one computer while limited to one process

Figure 4.39 shows the runtime of the Illimine PrefixSpan implementation on Windows® in comparison with my algorithm. The three runtimes of my algorithm are using four computers, the runtime using only one computer, and the runtime limited to a single process on one computer. The runtimes of my algorithm are lower than PrefixSpan when using multiple computers and when using multiple processes on one computer. The runtime of PrefixSpan is lower when my algorithm is limited to one process on one computer because PrefixSpan performs recursive projections of the database rather than enumerating each row. However, it is important to note that my algorithm is specifically designed for distributed- and shared-memory computers, rather than a single computer using only one process. In this experiment, when I use the multiple computers and processes that I designed my algorithm for, it achieves a lower runtime than PrefixSpan.

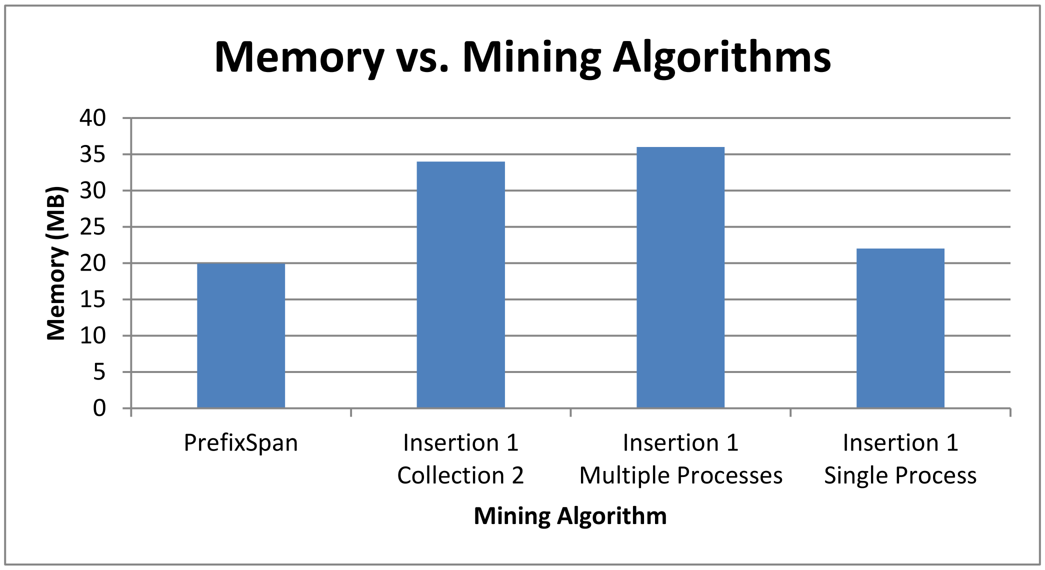


Figure 4.40: The memory usage of PrefixSpan in comparison to the runtime of my algorithm using four heterogeneous computers, one computer with multiple processes, and one computer while limited to one process

Figure 4.40 shows the memory requirements of PrefixSpan compared to those of my algorithm. PrefixSpan uses near the same amount of memory as my algorithm running in one process. It captures the input database to memory and stores two levels of database projections at a time. My algorithm stores the entire result set as well as the insertion list. Running my algorithm with multiple processes increases its memory usage, as it requires multiple copies of the insertion list. The reduction in memory usage when using four computers is due to my algorithm assigning one OpenMP process to receiving communication. The communicating process does not need an insertion list.

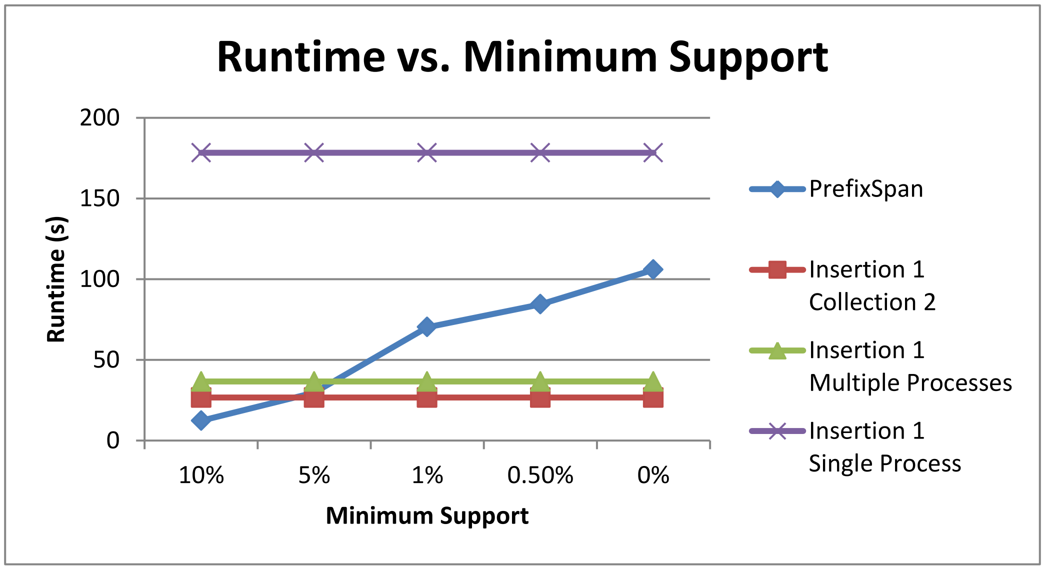


Figure 4.41: The runtime of PrefixSpan as the minimum support threshold decreases, compared to my algorithm

Figures 4.41 and 4.42 show the runtime and memory usage of PrefixSpan mining the synthetic dataset as the minimum support threshold decreases, again in comparison with my algorithm. The runtime of PrefixSpan increases as the minimum support threshold drops, while the runtimes of my algorithm remain the same. Again, my algorithm is specifically designed for distributed- and shared-memory computers. It cannot take advantage of my algorithmic design when limited to a single process on a single computer. When not limited to a single process, my algorithm requires less runtime than PrefixSpan when the minimum support threshold drops below 5%. When the minimum support threshold reaches 0%, my algorithm using multiple processes on one computer, and multiple processes on four computers, requires less than half

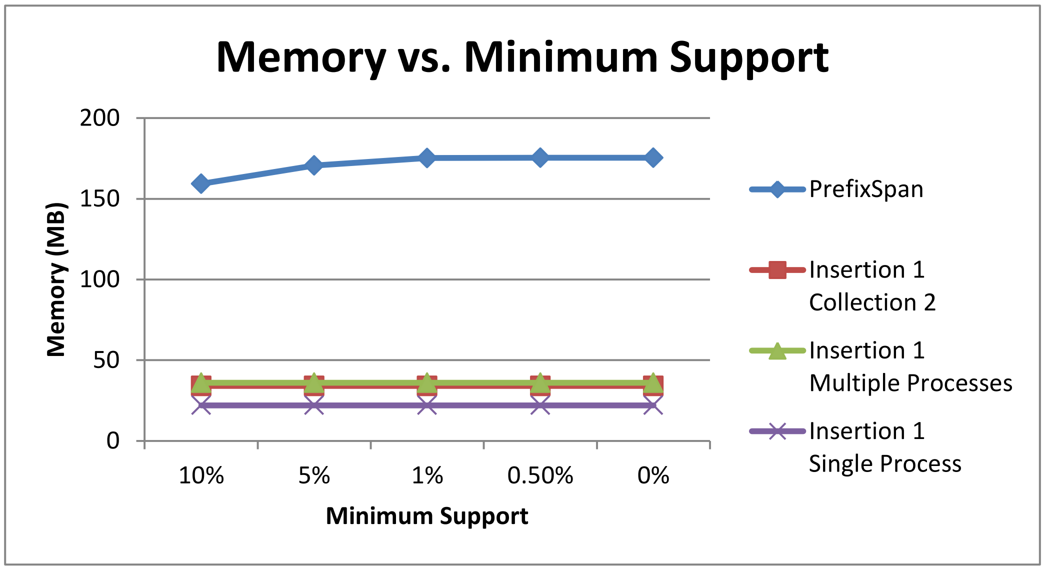


Figure 4.42: The memory usage of PrefixSpan as the minimum support threshold decreased, compared to my algorithm

the runtime of PrefixSpan. Interestingly, even though PrefixSpan only has to keep two levels of projected datasets in memory at any time, it requires far more memory than my algorithm as the minimum support threshold decreases. It has to keep more rows in each projected dataset as the number of rows in the input database increases.

I have compared the results of the PrefixSpan algorithm to the results of running my algorithm on the same dataset. The results match, showing the correctness of my algorithm.

4.4 Summary

Experiment 4.3.1 shows that my algorithm scales predictably when the number of processors increases, splitting the required runtime evenly between processors, and my greedy approach for reading input on each processor results in a balanced load.

It also shows that Hyper-Threading is beneficial when using all available processors and thread units, but has unpredictable performance due to the operating systems scheduling decisions when using a subset of available resources.

Experiment 4.3.2 shows that as the number of rows increases, the runtime of my algorithm increases linearly. The runtime is the time to mine one row, times the number of rows. Experiment 4.3.3 shows that as the row length increases, the runtime of my algorithm increases exponentially. Each additional item doubles the runtime.

Experiment 4.3.4 shows that the per-node collection cost of using MPI broadcast and reductions is much higher than the cost of using a file stream. Collection mode 1 performs slightly faster than collection mode 2 when it has to collect few nodes, but collection mode 2 performs much faster when many nodes are frequent. Experiment 4.3.5 shows that my load balancing method gives a reasonably balanced load, with an average difference between portion size percentages of 2.5, with no overhead when using my algorithm to perform data mining.

Experiment 4.3.6 shows that, for collection mode 2, increasing the number of homogeneous distributed-memory computers used divides the runtime evenly between computers. On heterogeneous distributed-memory computers, the runtime is the time to run the dataset assigned to any computer by the load balancer. For collection mode 1, as the number of computers increases, the cost of the MPI broadcast and reduction operations also increases, limiting the scalability of the algorithm.

Experiment 4.3.7 shows that serial frequent-sequence mining algorithms, such as PrefixSpan, can perform faster than my algorithm if my algorithm is limited to one process on one computer. This is expected, since I designed my algorithm to run on multiple processes on multiple computers. PrefixSpan captures the database, and then performs recursive projections of the database, which reduces the input cost per row. This performance benefit comes at the cost of a large increase in memory usage when the database contains a high number of rows. My algorithm performs faster when the minimum support values are low, and when it makes use of all available resources such as processors, processor cores, and thread units. This makes my algorithm a good choice when it can make use of multiple processing resources or when memory is limited.

Combining my results, insertion mode 1 performs faster than insertion mode 2, but requires more memory. Collection mode 1 performs faster than collection mode 2 only when the number of nodes it has to collect is low. Collection mode 2 requires less runtime when many nodes are frequent, and is more scalable, since it does not require MPI broadcast and reduction operations. It is possible to predict the time required to mine a dataset with collection mode 2 using previous runtimes from the mining computers, such as those from my load balancing mechanism. The parameters this prediction requires are the number of available computers, the number of rows in the dataset, and the average number of items in each row.

For instance, it takes 293 seconds for one of the bird computers to mine the repeat dataset with ten thousand rows of 20 items. Based on Equation (4.3), it would take 9 seconds for this computer to mine a repeat dataset with ten thousand rows of 15 items using insertion mode 1 and collection mode 2. Running this test shows that this estimate is accurate, with an actual runtime of 7 seconds. Applying Equation (4.1) for the number of computers to the 9-second estimate results in an estimated 10 computers required to reduce the runtime below 1 second and match the time to read the input database. Running the test on this dataset shows that it takes at least 8 computers before the average runtime is below one second, and 11 computers before the average runtime matches the time to read the input file, so this estimate is again accurate.

Due to the communication cost of collection mode 1, its runtimes are harder to predict, as it would require an estimate of the number of nodes that it will collect. Further experimentation shows that collection mode 1 can mine a dataset with ten thousand rows of 10 items in the time it takes to read the dataset using 2 to 7 computers, with the communication cost increasing the runtime past 7 computers. Collection mode 2 can mine this dataset in the time it takes to read the input file using at least 2 computers. Both collection modes can read the dataset with ten thousand rows of 5 items in the time it takes to read the input dataset using any number of computers.

In summary, my algorithm meets my goal of mining the entire database in near the time it takes to read the input database when there are enough computers available to do so. When the central database becomes a bottleneck, the difference between the runtime and the read only time is the time my algorithm spends mining the last input row, and the time spent collecting results. My algorithm reduces the number of database rows enumerated by each computer, and each processor or thread unit within each computer, by spreading the work over all available resources. There are enough computers to mine in near the time it takes to read the input if the expected time to mine one row is near the time to read a row. Dividing the expected runtime by the total number of rows gives a prediction of the time to mine one row.

If a computer is not constrained by memory, does not have multiple mining resources available, or the user only wants to find very frequent sequences, PrefixSpan performs faster than my algorithm. If the user is looking for an algorithm capable of mining from a data stream, that is capable of incremental mining, or that they can add, remove, or modify constraint values of without re-mining, my algorithm could be extended to support these needs. It also performs faster than PrefixSpan when using multiple OpenMP processes, multiple distributed-memory computers, or a combination of both.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The goal of my research was to create a frequent-sequence mining algorithm that performs one scan of a central database, and completes the mining process when it completes this scan. If I can force the central database to be a performance bottleneck in this algorithm, the runtime of my algorithm will be the time to mine the database, plus some relatively small extra time to finish mining the last sequence and collect the results. Further, there were four key questions I sought to answer, as I presented in Section 1.6.

1. "Can I mine frequent-sequences on a distributed memory computer using only one scan of the original database?"

   I designed two frequent-sequence mining algorithms that require one scan of the original database, one in Section 3.1.1, and the other in Section 3.1.2. I achieved this by having my algorithms completely mine each row from the input database as they read it. Both perform the mining process without requiring context or information from the rest of the database by enumerating all sub-sequences from each input row. In contrast, existing algorithms that do not perform multiple scans of the database perform multiple scans of a transformation of the database, such as a CanTree or CP-tree that I reviewed in Section 1.4.

2. "Can I avoid transforming the database and scanning said transformed database multiple times?"

   Since my frequent-sequence mining algorithms in Section 3.1 enumerate all sub-sequences directly from the input dataset, they do not need to capture the contents of the database to an intermediate structure. Existing algorithms that do not capture and transform the database perform multiple scans of the original database. For example, PrefixSpan (Section 2.3.1) or the Apriori algorithms (Section 1.3) mine all sequences of length k on the k^{th} scan of the database, all sequences of length $k+1$ on the $(k+1)^{st}$ scan of the database, and so on. I have not found another algorithm that does not perform multiple scans of either the original database or a transformation of it in literature.

3. "Can I limit the amount of communication between computers so that my algorithm scales well?"

   In Experiment 4.3.6, I found that the frequent-sequence mining algorithms that I created in section 3.1 are successful in keeping the amount of communication low, and are scalable in regards to the number of computers. I also found that of my results collection modes, collection mode 2, which I created in Section 3.2.2, scales better than collection mode 1, which I created in Section 3.2.1. Even though collection mode 1 reduces the number of nodes collected in comparison with collection mode 2, and moves counts from memory to memory rather than saving them to disk as an intermediate step, the communication cost of MPI broadcast and reduction operations is higher than that of writing to a file stream. This is since broadcast and reduction operations require all computers to act in sync, whereas when serializing to a file stream, each computer sends independently.

4. "Can I make use of multiple processors, cores, and thread units to reduce the runtime of my algorithm?"

   The frequent-sequence mining algorithms I created in Section 3.1 take advantage of multiple processors, cores, and thread units on each computer. They do this using the Hybrid Overlap model for combining MPI, which handles communication between distributed-memory computers, with OpenMP, which manages the shared-memory processes within each computer, as I describe in Section 3.1.3. Using the Hybrid Overlap model allows mining computers to flag sequences as potentially frequent while mining for collection mode 1, and allows the collecting computer to switch processors from mining to collecting using collection mode 2. Experiment 4.3.2 measures the performance of my frequent-sequence mining algorithms using multiple processor cores and thread units. Using multiple cores performs as expected, dividing the runtime using one core by the number of available cores. Using multiple thread units does not improve the performance as much as using multiple cores. It can cause the runtime not to scale as expected, depending on if the operating system schedules threads to separate cores, or uses multiple thread units on one core. When my algorithm uses all cores and thread units, so that scheduling does not make a difference, the performance improves and behaves predictably in all tests.

Ultimately, using insertion mode 1, my insertion list, and collection mode 2, serializing local trees, I was able to achieve my goal of mining in close to the time it takes to read the input dataset on heterogeneous computers, when the number of computers available is great enough to cause a performance bottleneck at the central database. The runtime of my algorithm is the time to read the database, plus the time to enumerate the last row of the database, and the time to collect results. Running my algorithm on homogeneous computers may require more or less computers than running it on heterogeneous computers, depending on the relative performance of each computer. My load balancing method provides a reasonably balanced load with no overhead during the execution of my algorithm, so the predicted runtime on a heterogeneous system is the predicted time of any computer in said system, using the input assigned by the load balancer.

Existing distributed-memory frequent-sequence mining algorithms are modifications of serial algorithms. I have designed my algorithm specifically for distributed-memory computers, rather than modifying an existing algorithm, so that it does not suffer as badly from the communication constraints the modified algorithms encounter. My algorithm meets the requirements I defined for the ideal distributed-memory frequent-sequence framework. Using collection mode 2, my serialization-based results collection method, it also matches the structure of the ideal framework I presented in section 2.4. When there are not multiple computing resources available, I have found that PrefixSpan performs faster than my algorithm, but may require more memory.

To summarize, my contribution is the proposal of a single-scan sequential mining algorithm. Of the components of this algorithm, the sub-sequence enumeration method is not only applicable to mining from a static database, as I have presented in this thesis, but can also be applied to mining from dynamic data streams. For instance, in real life applications, huge volumes of streaming data can be generated by wireless sensors. This data comes in at such a rapid rate and large quantity that users do not have the luxury of scanning the data stream multiple times. It is important to have a method to mine a sequence in a single scan. Another component of my algorithm, my load balancing method, could also be applied to mining from data streams. It knows the relative performance of each computer before they begin mining, so it could determine the relative rates at which to send data to distributed computers.

5.2 Future Work

My research provides opportunities for future work. By not performing any pruning during the mining process, my algorithm has many additional benefits, of which future work can take advantage. Examples of these are changing the values of constraints, adding or removing constraints, and adding data to the dataset, all of which my algorithm could perform without re-mining data. To change values of, add, or remove constraints, my algorithm would iterate through the results tree, and only return those sequences that satisfy the new constraints. To add new data to previous results, my algorithm could have the collecting computer serialize and save its final tree when it completes mining, and then load this serialized tree while it is collecting distributed serialized trees.

The results of Experiment 4.3.6, which showed that the computer collection mode 2 stores serialized trees on becomes saturated when 15 or more computers are saving to it, suggests a second opportunity for future work. One possible future improvement of my algorithm is adding the ability to scale the number of computers to which it saves serialized trees. Increasing the number of these computers would prevent my algorithm from saturating them with saves, and avoid them becoming a performance bottleneck and limiting scalability in regards to the number of computers used.

A third area of potential future work uses the ability to predict the required runtime of a dataset. For datasets with a large number of items per row, the time to insert each row will be high, and there may not be sufficient computers to mine in the time it takes to read the input dataset. Modifying my algorithm to capture the database before mining may perform better in this case. It would insert all of the input sequences into a tree, and then combine my depth-first and insertion list approaches to enumerate all sub-sequences from the tree. The modification would use the depth-first stack to keep track of the next sibling of each node to add to the insertion list, and the insertion list would enumerate all sub-sequences. Cases such as the repeat datasets would be the best case for this algorithm, since it would combine all of the input rows into one. This would still provide all of the additional benefits of my algorithm, and maintain the scalability, but would leave the central database idle during the mining process.

Other possible future work would include using a faster structure than an ordered list to track child nodes. Giving each tree node an existential probability would add support for uncertain mining, and adding an additional node type to differentiate between ordered and unordered nodes would add support for mining sequences of unordered itemsets. Adding the option to sort input sequences into a pre-defined order before enumeration would let my algorithm perform frequent-itemset mining rather than frequent-sequence mining.

Bibliography

[AIS93] R. Agrawal, T. Imieliński, 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*, pages 207–216, New York, NY, USA, 1993. ACM.

[ALWW09] C. Aggarwal, Y. Li, J. Wang, and J. Wang. Frequent pattern mining with uncertain data. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 29–38, New York, NY, USA, 2009. ACM.

[Arg] Argonne National Laboratory. MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE)*, pages 3–14, Los Alamitos, CA, USA, 1995. IEEE Computer Society.

[AS99] R. Agrawal and R. Srikant. IBM quest synthetic data generation code. Technical report, IBM Almaden Research Center, San Jose, CA, 1999. http://www.almaden.ibm.com/cs/quest/syndata.html.

[CHHP05] S. Cong, J. Han, J. Hoeflinger, and D. Padua. A sampling-based framework for parallel data mining. In *Proceedings of the ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming*, pages 255–265, New York, NY, USA, 2005. ACM.

[CHP05] S. Cong, J. Han, and D. Padua. Parallel mining of closed sequential patterns. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 562–567, New York, NY, USA, 2005. ACM.

[CJv07] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, Cambridge, MA, 2007.

[CMDAN08] M. Curtis-Maury, X. Ding, C. Antonopoulos, and D. Nikolopoulos. An evaluation of OpenMP on current and emerging multithreaded/multicore processors. In M. Mueller, B. Chapman, B. de Supinski, A. Malony, and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin / Heidelberg, Berlin, Germany, 2008.

[EZ06] M. El-Hajj and O.R. Zaiane. Parallel leap: large-scale maximal pattern mining in a distributed environment. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS 2006)*, pages 135–142, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[FZZ$^+$05] Y.-W. Fang, X.-B. Zhao, G.-P. Zhang, Y. Wang, Y. Sun, and Y.-F. Zhang. Study on algorithms of parallel and distributed data mining calculating process. In *Proceedings of the 2005 International Conference on Machine Learning and Cybernetics (ICMLC)*, volume 4, pages 2084–2089, Los Alamitos, CA, USA, Aug 2005. IEEE Computer Society.

[GABC08] I. Galindo, F. Almeida, and J. Badía-Contelles. Dynamic load balancing on dedicated heterogeneous systems. In A. Lastovetsky, T. Kechadi, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 64–74. Springer Berlin / Heidelberg, Berlin, Germany, 2008.

[Gho06] S. Ghosh. *Distributed Systems: an Algorithmic Approach*, volume 13 of *Chapman & Hall/CRC computer and information science series.* CRC Press, Boca Raton, FL, 2006.

[GPM$^+$09] G. Giarratana, M. Pizzera, M. Masseroli, E. Medico, and P.L. Lanzi. Data mining techniques for the identification of genes with expression levels related to breast cancer prognosis. In *Proceedings of the Ninth IEEE International Conference on Bioinformatics and BioEngineering (BIBE 2009)*, pages 295–300, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[HCC$^+$06] J. Han, H. Cheng, S. Cong, X. Li, H. Liu, Z. Shao, J. Wang, D. Xin, X. Yan, and X. Yin. Illimine. http://illimine.cs.uiuc.edu/about/contributors.php, 2006.

[HSJ$^+$08] L. Hualei, L. Shukuan, Q. Jianzhong, Y. Ge, and L. Kaifu. An efficient frequent pattern mining algorithm for data stream. In *Proceedings of the 2008 International Conference on Intelligent Computation Technology* *and Automation (ICICTA)*, pages 757–761, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[Int] Intel® Corporation. Intel® Hyper-Threading Technology (Intel® HT Technology). http://www.intel.com/technology/platform-technology/hyper-threading/index.htm.

[Ker88] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[KHK10] M. Kholghi, H. Hassanzadeh, and M. Keyvanpour. Classification and evaluation of data mining techniques for data stream requirements. In *Proceedings of the 2010 International Symposium on Computer Communication Control and Automation (3CA)*, pages 474–478, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[LB08] C. Leung and D. Brajczuk. Efficient mining of frequent itemsets from data streams. In *Proceedings of the 25th British National Conference on Databases (BNCOD)*, pages 2–14, Berlin, Germany, 2008. Springer-Verlag.

[LB09a] C. Leung and D. Brajczuk. Efficient algorithms for the mining of constrained frequent patterns from uncertain data. *SIGKDD Explorations*, 11(2):123–130, Dec 2009.

[LB09b] C. Leung and D. Brajczuk. Mining uncertain data for constrained frequent sets. In *Proceedings of the 13th International Database Engineering & Applications Symposium (IDEAS 2009)*, pages 109–120, New York, NY, USA, 2009. ACM.

[LB10] C. Leung and D. Brajczuk. uCFS$_2$: an enhanced system that mines uncertain data for constrained frequent sets. In *Proceedings of the 14th International Database Engineering & Applications Symposium (IDEAS 2010)*, pages 32–37, New York, NY, USA, 2010. ACM.

[LBY08] C. Leung, D. Brajczuk, and J. Yu. Efficient algorithms for stream mining of constrained frequent patterns in a limited memory environment. In *Proceedings of the 12th International Database Engineering & Applications Symposium (IDEAS 2008)*, pages 189–198, New York, NY, USA, 2008. ACM.

[LHB10] C. Leung, B. Hao, and D. Brajczuk. Mining uncertain data for frequent itemsets that satisfy aggregate constraints. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*, pages 1034–1038, New York, NY, USA, 2010. ACM.

[LKLH07] C. Leung, Q. Khan, Z. Li, and T. Hoque. CanTree: A canonical-order tree for incremental frequent-pattern mining. *An International Journal on Knowledge and Information Systems (KAIS)*, 11(3):287–311, Apr 2007.

[LLHC08] C. Liao, Z. Liu, L. Huang, and B. Chapman. Evaluating OpenMP on chip MultiThreading platforms. In M. Mueller, B. Chapman, B. de Supinski, A. Malony, and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 178–190. Springer Berlin / Heidelberg, Berlin, Germany, 2008.

[LLN00] L. V. S. Lakshmanan, C. K.-S. Leung, and R. T. Ng. The segment support map: Scalable mining of frequent itemsets. *SIGKDD Explorations*, 2(2):21–27, Dec 2000.

[LLW09] X. Li, H. Li, and Z. Wu. Model-driven data mining in the oil & gas exploration and production. In *Proceedings of the Second International Symposium on Knowledge Acquisition and Modeling (KAM 2009)*, volume 3, pages 20–24, Los Alamitos, CA, USA, Dec 2009. IEEE Computer Society.

[LMB08] C. Leung, M. Mateo, and D. Brajczuk. A tree-based approach for frequent pattern mining from uncertain data. In *Proceedings of the 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2008)*, pages 653–661, Berlin, Germany, 2008. Springer Berlin / Heidelberg.

[LVT$^+$10] P.J.G Lisboa, A. Vellido, R. Tagliaferri, F. Napolitano, M. Ceccarelli, J.D. Martin-Guerrero, and E. Biganzoli. Data mining in cancer research [application notes]. *IEEE Computational Intelligence Magazine*, 5(1):14–18, Feb 2010.

[LY09] B. Liu and Y. Yang. Mining algorithm for unordered embedded frequent subtree. *Computer Engineering*, 35(3):51–53, 2009.

[Mes] Message Passing Interface Forum. MPI-Forum website. http://www.mpi-forum.org/.

[Opea] Open MPI Project. Open MPI. http://www.open-mpi.org/.

[Opeb] OpenMP Architecture Review Board. OpenMP Website. http://www.openmp.org/.

[PD11] P. Paranjape-Voditel and U. Deshpande. An association rule mining based stock market recommender system. In *Proceedings of the Second International Conference on Emerging Applications of Information Technology (EAIT 2011)*, pages 21–24, Los Alamitos, CA, USA, Feb 2011. IEEE Computer Society.

[PHM+04] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The PrefixSpan approach. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(11):1424–1440, Nov 2004.

[RHJ09] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. pages 427–436, 2009.

[RS10] J. Renjit and K. Shunmuganathan. Mining the data from distributed database using an improved mining algorithm. *International Journal of Computer Science and Information Security*, 7(3):116–121, 2010.

[SLC08] 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)*, pages 1800–1805, Los Alamitos, CA, USA, April 2008. IEEE Computer Society.

[SS08] P. Sujnil and V. Saravanan. Hash partitioned Apriori in parallel and distributed data mining environment with dynamic data allocation approach. In *Proceedings of the International Conference on Computer Science and Information Technology (ICCSIT 2008)*, pages 481–485, Los Alamitos, CA, USA, Sept 2008. IEEE Computer Society.

[SSM10] A. Shamim, M.U. Shaikh, and S.U.R. Malik. Intelligent data mining in autonomous heterogeneous distributed bio databases. In *Proceedings of the Second International Conference on Computer Engineering and Applications (ICCEA 2010)*, pages 6–10, Los Alamitos, CA, USA, March 2010. IEEE Computer Society.

[STL+05] C. She, J. Tang, L. Li, H. Wang, and Z. Fan. An improved parallel algorithm for sequence mining. In *Proceedings of the 2005 IEEE International Conference on Mechatronics and Automation (ICMA), Volume 4*, pages 1692–1696, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[TAJ09] S. Tanbeer, C. Ahmed, and B.-S. Jeong. Parallel and distributed frequent pattern mining in large databases. In *Proceedings of the 11th*

IEEE International Conference on High Performance Computing and Communications (HPCC 2009), pages 407–414, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[TAJL09] S. Tanbeer, C. Ahmed, B.-S. Jeong, and Y.-K. Lee. Efficient single-pass frequent pattern mining using a prefix-tree. *Information Sciences*, 179(5):559–583, Feb 2009.

[Thu09] B. Thuraisinghami. Data mining for malicious code detection and security applications. In *Proceedings of the IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technologies (WI-IAT 2009)*, pages 6–7, Los Alamitos, CA, USA, Sept 2009. IEEE Computer Society.

[UCL08] G. Utrera, J. Corbalán, and J. Labarta. Dynamic load balancing in mpi jobs. In J. Labarta, K. Joe, and T. Sato, editors, *High-Performance Computing*, volume 4759 of *Lecture Notes in Computer Science*, pages 117–129. Springer Berlin / Heidelberg, Berlin, Germany, 2008.

[WFC09] C.-H. Wu, K. Fang, and T.-C. Chen. Applying data mining for prostate cancer. In *Proceedings of the International Conference on New Trends in Information and Service Science (NISS 2009)*, pages 1063–1065, Gyeongju, South Korea, 2009.

[YCM11] Y. Yuguang, W. Chunyan, and L. Min. Application of the model multi based on apriori algorithm in supporting system of medical decision. In *Proceedings of the Third International Conference on Measuring Technology and Mechatronics Automation (ICMTMA 2011)*, volume 1, pages 566–569, Los Alamitos, CA, USA, Jan 2011. IEEE Computer Society.

[YFP$^+$10] T. Young, M. Fehskens, P. Pujara, M. Burger, and G. Edwards. Utilizing data mining to influence maintenance actions. In *Proceedings of the 2010 IEEE Autotestcon*, pages 1–5, Los Alamitos, CA, USA, Sept 2010. IEEE Computer Society.

[ZLLW09] Z.-P. Zhang, Y. Li, Z.-J. Lin, and A.-J. Wang. Frequent itemsets mining algorithm based on index array. *Application Research of Computers*, 26(1):44–46, Jan 2009.