PARALLEL CLOSET+ ALGORITHM FOR FINDING FREQUENT CLOSED ITEMSETS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TAYFUN ŞEN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2009

Approval of the thesis

"PARALLEL CLOSET+ ALGORITHM FOR FINDING FREQUENT CLOSED ITEMSETS"

submitted by **Tayfun Şen** in partial fullfillment of the requirements for the degree of **Master of Science in Computer Engineering** by,

Prof. Dr. Canan Özgen _____
Dean, **Graduate School of Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit _____
Head of Department, **Computer Engineering**

Dr. Cevat Şener _____
Supervisor, **Computer Engineering Department, METU**

Prof. Dr. İsmail Hakkı Toroslu _____
Co-supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Asst. Prof. Dr. Tolga Can _____
Computer Engineering Department, METU

Dr. Cevat Şener _____
Computer Engineering Department, METU

Asst. Prof. Dr. Pınar Şenkul _____
Computer Engineering Department, METU

Asst. Prof. Dr. Tuğba Taşkaya Temizel _____
Information Systems Department, METU

Dr. Onur Tolga Şehitoğlu _____
Computer Engineering Department, METU

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Tayfun Şen

Signature :

ABSTRACT

PARALLEL CLOSET+ ALGORITHM FOR FINDING FREQUENT CLOSED ITEMSETS

Şen, Tayfun

M.S., Department of Computer Engineering

Supervisor: Dr. Cevat Şener

Co-Supervisor: Prof. Dr. İsmail Hakkı Toroslu

July 2009, 80 pages

Data mining is proving itself to be a very important field as the data available is increasing exponentially, thanks to first computerization and now internetization. On the other hand, cluster computing systems made up of commodity hardware are becoming widespread, along with the multicore processor architectures. This high computing power is synthesized with data mining to process huge amounts of data and to reach information and knowledge.

Frequent itemset mining is a special subtopic of data mining because it is an integral part of many types of data mining tasks. Often this task is a prerequisite for many other data mining algorithms, most notably algorithms in the association rule mining area. For this reason, it is studied heavily in the literature.

In this thesis, a parallel implementation of CLOSET+, a frequent closed itemset mining algorithm, is presented. The CLOSET+ algorithm has been modified to run on multiple processors simultaneously, in order to obtain results faster. Open MPI and Boost libraries have been used for the communication between different processes and the program has been tested on different inputs and parameters. Experimental results show that the algorithm exhibits high speedup and efficiency for dense data when the support value is higher than a determined value. Proposed parallel algorithm could prove to be useful for

application areas where fast response is needed for low to medium number of frequent closed itemsets. A particular application area is the Web where online applications have similar requirements.

Keywords: Frequent Itemset Mining, Data Mining, Parallel Computing, CLOSET+, FP-tree

ÖZ

KOŞUT CLOSET+ ALGORİTMASI İLE SIK KAPALI NESNE KÜMELERİNİN BULUNMASI

Şen, Tayfun

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Dr. Cevat Şener

Ortak Tez Yöneticisi: Prof. Dr. İsmail Hakkı Toroslu

Temmuz 2009, 80 sayfa

Verinin, önce bilgisayarlaşma, sonra da internetleşme sayesinde çok yüksek bir hızda arttığı günümüzde veri madenciliği çok önemli bir alan olarak öne çıkmaktadır. Öte yandan, ucuz donanım kullanılarak kurulan küme hesaplama sistemleri ve çok çekirdekli işlemciler yaygınlaşmaktadırlar. Bu yüksek hesaplama gücü veri madenciliği ile birleştirilerek büyük miktardaki verinin işlenmesi ve bilgiye ulaşılması amaçlanmaktadır.

Sık nesne kümeleri madenciliği, birçok veri madenciliği alanının gerekli bir parçası olması nedeniyle önemli bir alt alandır. Sık nesne kümeleri madenciliği başta ilişkisel kural madenciliği alanında olmak üzere birçok veri madenciliği algoritması için gerekli bir ön iştir. Bu nedenle, bu alt alan geçmişte sıkça araştırılmıştır.

Bu tezde, sık kapalı nesne kümelerinin bulunmasında kullanılan bir algoritma olan CLOSET+ algoritmasının koşut bir sürümü sunulacaktır. CLOSET+ algoritması, birden fazla işlemcide koşut bir şekilde çalışacak şekilde değiştirilmiş, böylece sonuçlara daha hızlı ulaşılması amaçlanmıştır. Farklı işlemlerin iletişimi için Open MPI ve Boost kütüphaneleri kullanılmış, program farklı veriler ve değişkenler açısından incelenmiştir. Deneysel sonuçlar programın yoğun veritabanlarında ve belirli bir destek değerinden yüksek durumlarda yüksek bir hızlanma ve verimlilik ile çalıştığını göstermektedir. Bu tezde

anlatılan koşut algoritma düşük ile orta sayıdaki kapalı sık nesne kümelerinin hızlı bir şekilde bulunmasının gerektiği uygulama alanlarında faydalı olabilir. Örnek bir alan çevrimiçi uygulamaların benzer gerekliliklere sahip olduğu Genel Ağdır.

Anahtar Kelimeler: Sık Nesne Kümeleri Madenciliği, Veri Madenciliği, Koşut Hesaplama, CLOSET+, FP-tree

ACKNOWLEDGMENTS

I would like to first of all thank my supervisor, Dr. Cevat Şener for his warm encouragement and guidance throughout my research work. He was the one who supported me when I thought of letting go. I also have to thank my co-supervisor, Prof. Dr. İsmail Hakkı Toroslu for his guidance and precious advise regarding techniques and algorithms I was to choose.

I would like to express my most heartfelt gratitude to my family, who put up with all my complaints and rants about everything under the sun in these three years of higher education. Mum, dad and my two sisters - this would not have happened without you.

Finally, I'm indebted to Imai-san (domo arigato gozaimasu), Semih, Gladys, Tuba, Ayşenur, Olduz, Zeki, Melih, Tamer, Tufan, Metin, Alptekin & Joanna and many others, who have each touched my life in way or another. My life has been more interesting thanks to you.

I have been partially funded by the Turkish Scientific and Technical Council (TÜBİTAK) BİDEB 2210 National Graduate Scholarship Programme. Thanks goes to many educational programmes I have had chance to attend to provided by TÜBİTAK and Computer Engineering Department (CENG) at the Middle East Technical University (METU). All the tests were run on the grid computing infrastructure provided by CENG and TÜBİTAK.

Parts of the work in this thesis has been published in the proceedings of the BAŞARIM'09 conference on high performance computing.

As a side note, this thesis has been typeset using LaTeX, edited using the great Vim editor and graphics generated using gnuplot software all running on Debian GNU/Linux distribution. Thanks to the Open Source community for these excellent products!

The whole source code for the project described in this thesis can be found at the Google Code SVN repository at: http://code.google.com/p/parcloset/ .

O'na, aileme ve hor görülmüş ülkeme...

Contents

List of Figures

FIGURES

List of Tables

TABLES

LIST OF ABBREVIATIONS

API Application Programming Interface

CERN European Organization for Nuclear Research

CPU Central Processing Unit

ddd Data Display Debugger

FIM Frequent Itemset Mining

gdb The GNU Debugger

IP Internet Protocol

KDD Knowledge Discovery in Databases

LHC Large Hadron Collider

MB Megabyte

MPI Message Passing Interface

MPMD Multiple Program Multiple Data

OpenMP Open Multiprocessing

OS Operating System

PB Petabyte

PC Personal Computer

PVM Parallel Virtual Machine

SMP Symmetric Multiprocessing

SPMD Single Program Multiple Data

TCP Transmission Control Protocol

UYBHM National High Performance Computing Center (Ulusal Yüksek Başarımlı Hesaplama Merkezi)

vim Vi IMproved, the text editor

Chapter 1

INTRODUCTION

In this chapter conducted research work will be discussed briefly. The reason and rationale behind data mining and why parallel computing dimension has been added will be discussed as the first topics. In the following section, the approach taken in our research will be explored. Some of the design decisions that had to be taken will be given briefly, their details left to the following chapters. Introduction chapter will be concluded with an overview and organization plan for the rest of this document.

1.1 Reason and Rationale

We are in what is called *The Information Age*. The full impact and our position in this Digital Revolution is not clear but it has proven to offer new challenges and opportunities. Information overload is one of those challenges. With the widespread use of computers and the Internet for anything and everything worth using, excess amount of data has been generated, often causing the individual to be confused. Data made accessible has risen unexpectedly fast in the last century and it has proven to be extremely difficult for the individual to process and extract meaningful information and leverage his knowledge in any field. In one specific example, the Large Hadron Collider at CERN is expected to generate data of size about 700 MB per second or 15 PB per year [4]. Humans are found that they cannot keep up with this new kind of world where data is abundant and information pollution is all too widespread.

Knowledge Discovery in Databases (KDD) is thought to be the answer to information pollution. KDD represents the techniques and procedures in getting to knowledge from raw data. The three basic steps in KDD are [34]:

1. Data is preprocessed to be normalized, noises removed and formatted so that it can be mined easily and on target.

2. Data mining step is executed to extract patterns and thus transform the data to information.

3. The last step in KDD is the interpretation and evaluation of the results. Thus, knowledge is formed.

In this thesis we are concerned with the data mining step in KDD. This step is the most technically challenging step that needs to be done in KDD. As such, most of the research work has focused itself on data mining techniques. Nevertheless, data preprocessing and interpretation of mined patterns are also vital steps which impact the correctness of the findings.

Data mining is the process and techniques for extracting patterns from data. Data mining techniques can process huge amounts of data and extract patterns that have specific desired properties. Humans are known to mine data intuitively and manually, albeit using less amount of data and at a much lower speed when compared to computers. Data mining uses the ever increasing processing power ubiquitous in today's computers to present data in an understandable way for humans. Extracting patterns from data has enabled companies and individuals to access information they need from a vast pool of resources. This has enabled advanced and more precise predictions, forecasting, situation analysis and decision support systems. It has enabled advanced research in many areas of science such as bioinformatics, genetics, physics and finance to name a few. Many of the technological advances we marvel at today use data mining techniques to extract information and arrive at the results.

After the computerization of many of the work flows in almost all industrial sectors, convergence to the Internet is becoming the norm. The boundaries between local desktop services and those that are provided remotely are becoming blurred. Many of the routine works normally done locally are being transferred to the so-called "cloud" [47], computations are done remotely and results sent for the local viewing to the user. Users don't know where their data is sent to, where the computations are done and most of the time, they don't care. So called "internetization" is gaining momentum as more and more people are connecting to the Internet for the first time, or by using mobile devices they are staying online for extended periods of time. Portable and always connected devices, much improved Internet infrastructure and initiatives such as One Laptop Per Child (OLPC) are helping

this convergence towards the Internet. As more and more people become online for longer periods of time, different services have proliferated catering to the newer needs of people. The success of Web 2.0 with collaborative web sites such as flickr, youtube, twitter and many social networking web sites such as facebook and myspace can be tracked down to this phenomena [67].

What effect does "internetization" have on data mining? As more and more people spend more and more of their time online, data generated on the Internet increases rapidly. One direct result can be seen in the number of websites search engines like Google crawl and index. Although there is no single authority keeping track of the number of web pages, it is anticipated that growth has been extraordinary. Netcraft March 2009 survey [11] estimates that about 225 million web pages exist today. The huge increase in data first thanks to "computerization" and now to "internetization" accelerated the need for faster and more accurate data mining techniques.

Grid computing environment is given as the solution for the high computational power requirements in data mining. Grid computing can simply be described as using many computers, which can be heterogeneous and geographically dispersed, to attack and solve problems concurrently. The huge computational power grid computing provides can be tracked down to its use of many computing elements simultaneously. This increases the rate at which data can be stored, read and processed.

To facilitate the computational power provided by grid computing environments, several methods have been proposed. Developments in this area include easy and seamless shared memory thread management using compiler directives [31], distributed memory message passing libraries [41], implicitly parallel languages [35], or even whole operating systems providing resource sharing capabilities to programs without forcing any extra work on behalf of the programmer [22].

We see that data and information "out there" is getting bigger all the time, while people want faster access to relevant information. This provides the motives for faster data mining algorithms. As multicore CPUs, terabyte scale hard disks and better all-round computers become commodity and distributed computer systems become cheaper, newer methods addressing data mining on the parallel scale becomes indispensable in achieving even more efficient implementations. Parallel programming practices have become widespread in many areas of science where performance is vital. Data mining is no exception and parallel programming offers new speed gains and efficiency in this field. In the next section we delve into more details of the work with the approach taken.

1.2 Approach

In this thesis, a new parallel version of a data mining algorithm called CLOSET+ [77] is described. CLOSET+ is a successful algorithm for finding frequent itemsets in databases. This fast algorithm has been revised to make use of more than one processor efficiently. The algorithm presented in this thesis is the first parallel CLOSET+ implementation to the best of our knowledge. The new parallel algorithm has been implemented using message passing techniques and it is able to run on shared-nothing architectures. During performance tests it is found that for dense datasets and for a range of support values, this new parallel algorithm performs quite efficiently.

CLOSET+ algorithm has been modified to include constructs that enable efficient parallel computations. The division of the job and also the relevant data, merging of intermediate results, creating new work packages according to the intermediate results and distributing this workload to computing elements and merging the final results are some of the problems addressed. The main CLOSET+ algorithm has been employed at each computing node for generating local computation results. More detailed descriptions of the algorithm is left for the following chapters.

The algorithm has been has been developed on a PC running Debian GNU/Linux operating system, using the C++ language. The reason C++ language has been chosen is many fold. The language has low-level features like manual memory operations and this brings flexibility for the programmer. The language has very fast execution which is what the whole aim of the research is. All the big MPI library implementations are written in C language, which can easily be integrated to C++ code. Having a widely praised and portable compiler such as GCC is also one of the merits of using C++. The reason for choosing C++ over C is because C++ provides more high level features. Object oriented programming support from C++ has enabled easier custom data structures. Tree and graph like structures designed and coded during implementation would have been quite complex and ugly, had C been chosen.

Combining object oriented programming methodologies of C++ with those of C API function calls from an MPI library would have been ugly and unmaintainable. To follow the object oriented programming guidelines, a C++ MPI library from Boost libraries has been used. Thanks to Boost libraries, MPI calls have been made using object oriented coding conventions.

One shortcoming of current MPI implementations with respect to object oriented

programming methodologies is that there is no way to send custom data types through MPI calls directly. The method of sending a custom data type is using MPI_Pack and MPI_Unpack functions to package whole data structure and unpack it at the receiver's end. This is often a tedious task that is prone to errors. To have a simpler method for sending custom data types, and following our object oriented programming guidelines as described earlier, another Boost library has been used. Boost serialization library offers easier way to serialize any data type, no matter how complex it is. The library also offers ready made implementations for many C++ constructs such as "string", "hash_map" and "vector" data types. Boost serialization library plays nicely with the MPI library from the same suite of libraries. Thus, one can directly send many complex data types like vectors, maps etc. from one node to another without having to worry about packing and unpacking operations. Sending custom data types also becomes very easy. Simply by extending your custom class so that it implements an interface, you can send objects easily.

Boost MPI library can use many C language MPI implementations on the background. For the implementation discussed in this thesis, Open MPI has been used. Open MPI is one of the popular and advanced MPI distributions. It is formed by merging of a couple of existing MPI projects and have borrowed ideas from many others. Flexible configuration options Open MPI offers has been a key factor in choosing this library. It is also installed by default in the grid computing environments we had access to.

Debugging of parallel programs is a tough topic at its best and it deserves a book of its own. For the project in this thesis, ddd (Data Display Debugger) has been used. The ddd is a graphical debugger which uses the gdb (gnu debugger) on the background. The most advantageous feature of ddd is its ability to display graph structures with ease. One can click on pointers to follow them and see memory locations mapped to the objects. All the regular debugger operations such as breakpointing, stepping, displaying variables etc. are available in a graphical user interface as well as a command line area which is directly tied to the gdb.

To complete the toolkit, vim has been used as the text editor of choice. Various plugins also complemented vim functionality, for example ctags utility has been used to jump from one source file to another one where the function under the cursor is defined. More details of the configuration is left to the following chapters.

A small work of data preprocessing has been done in the implementation, mainly of pragmatic reasons. In reading data sets, two different formats have been taken into consideration. The first format is a simple format which has each transaction on a line of

its own, and each items in the transactions given on this line. The second form of data is the one output by Quest IBM synthetic dataset generator. This dataset has different characteristics and our project has been implemented so that this format can be recognized as well. More information on data formats is be left to the following chapters.

Performance tests on the implementation show considerable speedups for certain data at a range of support levels. To be more specific, the algorithm and implementation works best with dense data sets and with medium or higher support levels. At these levels, speed up becomes as high as 85%. With sparse data sets, the speed up value drops, this phenomena is also seen with lower support values. With lower support values, single processor execution first becomes on par with the multiprocessor execution and later it becomes faster than the multiprocessor run.

The results show that for specific data sets this algorithm provides promising performance. This is important for many applications, such as web sites where users expect quick and correct responses. In such applications, the whole set of frequent itemsets generated using lower support values are not needed. In applications where fast results are desired, the algorithm discussed in this paper may be a suitable answer.

1.3 Scope of the Document

The rest of the thesis is organized as follows. Two basic subjects are data mining and parallel programming which are the building blocks for our research. They are presented in the next two chapters, respectively. In the data mining chapter, a top to bottom approach is taken. Topics start from the more general subject of knowledge discovery to the more specific area of frequent itemset mining. Also discussed in this chapter is a general view of the data mining field, current problems and research directions. In the following chapter, parallel programming methods is discussed including MPI, which has been used in our project. Chapter 4 presents the implementation, including the algorithm, solution framework and implementation details. Step by step explanation of the parallel algorithm is also presented in this chapter, aiming to easier express the algorithm employed. The following chapter, chapter 5, includes experimental results with various graphics and explanations regarding them. The final chapter in this document presents the concluding remarks.

Chapter 2

OVERVIEW OF DATA MINING

In this chapter, previous research on knowledge discovery and data mining will be presented. This overview starts with an introduction to the more general concept of Knowledge Discovery in Databases. Some notable uses of Knowledge Discovery in Databases are explored along with the future trends. The value added services like flickr interestingness, amazon suggestions and google flu trends will be articulated upon.

The section following KDD is about the more specialized topic of data mining. Data mining is often considered as an indispensable step in knowledge discovery. History of data mining research is provided and so it is hoped that the reader will be able to appreciate the developments in this relatively young research area. Recent data mining algorithms are explored and future trends elaborated upon. Several areas of data mining including classification, regression and prediction, clustering and the topic of research for this thesis, association rule mining are explored.

Serial and parallel frequent itemset mining algorithms will also be presented in this chapter. A brief overview of how data mining is done in parallel is presented for each algorithm. For many of the serial algorithms, their parallel counterparts, if existent, are discussed in the following section.

The chapter ends with a discussion of current problems associated with data mining and how these problems are addressed. Some solutions that have been proposed are discussed briefly.

This chapter, along with the next one describing parallel programming practices, sets the stage for the more detailed data mining content describing the performed research.

2.1 Knowledge Discovery in Databases

Knowledge Discovery in Databases is a term that has been coined in 1989 with the precursor workshop that would later become KDD conference series [63]. KDD and data mining terms are relatively new, but their applications had been developed much more earlier than the inception of these definitions. KDD is generally defined as the process of extracting implicit information and knowledge from data [34]. KDD helps humans make sense of huge amounts of data by mining patterns. Formally, KDD encompasses data preprocessing, data mining and results interpretation steps but the term "data mining" itself has been used synonymously with KDD by many practitioners in the field. In this thesis, the original meaning of KDD will be preserved, although no clear distinction will be made between these two terms.

As discussed in Section 1.1 in the Introduction Chapter, KDD grew out of necessity. The huge increase in the amount of data accumulated with the help of computerization created the need for newer methods which can extract information and knowledge. These methods were expected to mine huge data and extract information in a reasonable time. With the higher computer penetration and the Internet itself, the data need to be mined has increased dramatically. Performance expectations from data mining has also risen, in many data mining implementations designed for the web, the results are expected to be returned in seconds.

Before diving into data mining algorithms in the coming sections, it will be beneficial to assess some of the KDD applications and try to understand the big picture. Out of many important data mining implementations it is hard to choose a representative set. Some of the more novel, popular and influential applications have been google flu trends, amazon.com suggestions and flickr interestingness. In the remainder of this section each application will be analyzed.

Google flu trends is a recent project led by the Google Foundation [7]. In this project, Google engineers worked with the US Centers for Disease Control and Prevention to set up an early alert system for flu outbreaks. Search keywords submitted by people to the search engine is used in this project to find correlations with flu activity. Google search engine usage is mined and the used keywords, originating IP location, and other related data is used in finding the flu activity in different states. The simple idea behind this project is that during flu seasons people search for more flu related topics. By making use of this knowledge, the Google team were able to predict flu outbreaks two weeks before

Centers for Disease Control and Prevention which used conventional methods like clinical data and physician visits. The findings of this project has been published as an article in the Nature magazine [37].

Many of the earlier KDD applications had focused itself on market basket analysis. In this type of work, retail data is analyzed and patterns about shopping habits of customers are found. A very popular example is finding which items are bought together, like beer and baby diapers, and using this information to have these items in nearby shelves in the supermarket. In an internet based shopping center, the shelves become web pages, but the basic idea does not change. This is what is implemented by amazon.com suggestions. When you visit an item's web page on amazon system and add it to your cart, you are presented with similar items. These suggested items are what other customers who have had similar cart as yours have bought as well. Thus, virtual shelves are being connected so as to make it easier to buy related items. It is anticipated that this kind of innovative approach results in a considerable sales increase [66].

As the last KDD example, we will be looking at interestingness feature of flickr website. Flickr.com is one of the most popular image sharing sites on the Internet, it is claimed that flickr.com hosts 3 billion images as of November 2008 [1]. Flickr owns the most of its popularity to its successful community building features build on top of easy image management interface. Flickr has made interacting with other people's photos very easy, through the use of tagging, commenting, adding a photo to favorites, photography groups where similar minded people get together and many other features. A big part of flickr's success is attributed to its novel and accurate data mining methods and the value added services built on top of data mining. When flickr search is used, wide array of information like number of views and the people viewing the photo, number of favourites and individuals who added the photo to their favorites, different tags, comments about the photo, photo licences, social link from the current user to user in question etc. are all used and the most relevant photos are shown to the user. Interestingness [6] is another feature flickr has that deserves a mention on its own. Through the use of complex algorithms and computations, the most interesting photos are shown on a web page to the user. This list is updated continuously and it changes according to an internal algorithm. The details of the KDD algorithm employed by flickr is not disclosed in detail, such as it would be in an academic paper, but the method is nonetheless filed for patent at the US Patent and Trademark Office and one can view its description [10].

The previous cases chosen to exemplify KDD are by no means complete representations

of the diverse types of tasks that KDD has been employed to. The list is chosen to give an idea of some of the different usages of KDD. In fact, KDD can be employed anywhere where there are huge amounts of data. Information can be extracted and knowledge formed, creating the basis for value added services. In the next sections, more details of data mining algorithms are provided.

2.2 Data Mining

Data mining is the actual step in which patterns in data are found. This step is generally performed after some initial preprocessing of raw data. The data is modified, cleaned and normalized to make it suitable and easy to run mining algorithms on.

Data mining is an intensely multidisciplinary field [34, 45]. It enlists help from fields such as artificial intelligence, statistics, machine learning, pattern recognition, database systems, expert systems, parallel programming, high performance computing, information retrieval and signal processing among others. Due to the nature of the information extracted and the knowledge to be formed, data mining also makes use of diverse application areas such as finance, medicine, genetics, retail and many other industries where a domain expert brings *wisdom* into interpreting the results, making the transition from information to knowledge.

Data mining algorithms are usually divided into two, depending on the aim of the task. These two types of algorithms are "predictive" and "descriptive" algorithms. The names are self-explanatory; the predictive algorithms try to predict the future coarse of events, what will happen and how the behaviour will be. The descriptive algorithms on the other hand try to describe the situation, to summarize certain parts of data. There is a thin and blurry line between predictive and descriptive algorithms; in many cases a descriptive algorithm provides a way to compute predictions and vice versa. For example, association rule mining is generally classified as a descriptive algorithm but it can nonetheless be used to predict which items a user is most likely to buy, given his current basket. As such, this kind of clear distinction is not emphasized in this thesis.

Data mining tasks can further be divided into many categories. The major data mining categories are classification, clustering, regression and association rule mining.

Classification is the task of assigning objects to predetermined classes, according to many attributes they possess. For example, the task of assigning a category such as "gambling", "chat-messaging", "news-entertainment" etc. to a web page is a type of classi-

fication. Another well known example is deciding which mail falls into "spam" type and which ones are legitimate and normal [39]. A number of different techniques are used in classification, including the more popular decision trees, Bayesian classifiers, neural networks and nearest neighbour techniques.

Clustering is very similar to classification with the difference being that there are no predetermined classes in clustering. Objects that are similar according to a metric (a distance measure) are grouped together and clusters are formed. Clustering is useful when the number of classes and their substance are not known beforehand. This technique can be used in many areas where finding similar objects is desired. It can be used for finding people similar to you in social network analysis, finding similar items in the retail sector and for image segmentation, among others.

Regression techniques are used for modeling data. It is used in a variety of fields where predicting unknown values is the aim. Regression makes use of many statistical methods in building the best model. The most common uses for regression is for time-series data, sequential data where time and order is important. A popular example of its usage is in the financial sector; predicting the loan behaviour, credit score or credit limits of a person can all be solved using regression.

Association rule mining is the main research topic of this thesis, and so deserves a section on its own. In the next section, it is described in detail.

2.3 Association Rule Mining

Association rule mining and its subproblem, frequent itemset mining, is one of the most popular and well researched areas in data mining. It is used for finding relationships (or association rules, patterns) between different variables in the data. Association rule mining is largely made popular by Agrawal et al. in their seminal paper [18]. The notion of association rules was existent even before this paper (one of the earliest examples: [56]), although its application to market basket data was a novel idea. The Apriori algorithm for mining association rules [19] would become the dominant algorithm from its inception to early 2000s. The context and definition of association rule mining, as given by Agrawal et al. will now be given.

Thanks to the introduction of the bar code technology and computerization, businesses found themselves with huge amounts of transaction data. This data is generally made up of items bought together by customers and is called market basket data. Some of the

terms which will be used throughout this document will be defined now.

Let I be the set of items that are available for sale. An *itemset* $A = \{i_1, \cdots, i_k\}$ is a set of items defined over the whole item set I, and is called a *k-itemset* if it has k number of items.

A *transaction* is a single sale information and it consists of a transaction ID (TID) and all the items that are included in the sale. When a retail database is referred to, a list of transactions must be understood. An example database which can be obtained from a simple store is presented in Table 2.1.

Table 2.1: A Simple Store Database

TID	Basket contents
001	milk, butter
002	bread, butter
003	milk, butter, honey
004	bread, honey
005	milk, honey

After the acquisition of huge amounts of sale data, the next step was -naturally- the processing of it. With the processing step, organizations aimed to uncover hidden information from their database. Utilizing this information helped organizations in many areas, including better marketing strategies, inventory plans, item suggestions and shelf organization plans [66, 18].

Association rules provide information about relationships between different variables in data. They are sometimes described as if-then type of rules. This can better be understood with an example: $\{milk,\ butter \Rightarrow honey\}$ is a typical association rule discovered from a database such as the one shown in Table 2.1. What this rule means is that customers who buy milk and butter together also buy honey. In a rule such as $A \Rightarrow B$, the left hand side, A, is called the *antecedent* and the right hand side, B, is called the *consequent*. A and B in the rule are itemsets containing items existing in the database, and their intersection is the empty set.

This simple rule leads us to the following questions: How are the rules determined, and is there a granularity showing how confident we are about a rule? Indeed, these issues are valid and addressed in research. In order to filter out uninteresting associations, rules that have too few examples, several metrics have been proposed. The two important measures that are central to association rule mining are *support* and *confidence*.

Definition 2.1
Support value of an itemset A, denoted as $sup(A)$, is the proportion of the transactions which include all the items in the set A. Formally, this is represented as:

$$sup(A) = \frac{Number\ of\ transactions\ containing\ itemset\ A}{Total\ number\ of\ transactions}$$

This metric shows how frequent an itemset occurs in the database.

Support value is used to prune the search space mining algorithms will be run on. It is used with a minimum support value, also called support threshold, and items with support values lower than the set minimum are pruned. These items, by the definition of support, are not frequently observed in the database, and thus not taken into consideration. Itemsets satisfying the minimum support value are said to be frequent. As an example, a support value of 40% for an itemset such as $A = \{milk,\ butter\}$ simply means that two out of five transactions in the database contain these two items. From the retail point of view, it means two out of five sales included these items together.

After defining *support*, we can now move on to the confidence metric.

Definition 2.2
Confidence value of a rule such as $A \Rightarrow B$ is defined as:

$$conf(A \Rightarrow B) = \frac{sup(A \cup B)}{sup(A)}$$

Here the support value of the union of itemsets A and B is divided with the support value of only the itemset A. This gives us how likely it is that items in B exist given that items in A exist in a transaction.

Confidence metric is a little more controversial than the support metric. According to Fayyad, Piatetsky-Shapiro et al. [34], confidence is simply another metric among many others for discovering *interesting* patterns. Confidence is controversial because after using support metric to find frequent itemsets, one can use many available metrics such as lift,

conviction etc. in finding interesting association rules. Still, confidence is the original metric used by Agrawal et al. [18, 19] and it is one of the most widely known. It is also considered successful with certain types of data, and easier to understand. In this thesis, other metrics will not be described, only confidence will be explored. For more information and survey about different interestingness metrics, please refer to [72].

As in the support case, a minimum confidence value, also called confidence threshold, is used to prune those rules having less confidence value than the minimum. A high confidence value shows a high correlation between the antecedent and the consequent. Generally, association rules having high confidence values are desired because the high correlation value between items translates into higher probability of these items bought together. Recent research however, shows that patterns with high confidence values generally constitute common knowledge and are thus uninteresting. There exists research focusing on effectiveness of different metrics of interestingness, refer to previously cited work [72] for more information. As an example, association rule $\{milk,\ butter\} \Rightarrow \{honey\}$ with a confidence value of 50% shows that half of all sales that include both *milk* and *butter* also include *honey*.

Although examples and definitions have all been tailored for the retail sector, association rule mining is not limited to market basket analysis. In fact, it can be used to find associations in any kind of database and between a wide range of variables. Census data could be used to find demographic information (rules of the sort $\{age > 18,\ gender = male\} \Rightarrow \{earns > 10k\}$), financial data could be used to decide whether credit can be given to a customer, or medical data can be used to design decision support systems helping medical doctors in making informed diagnosis.

2.3.1 Apriori Algorithm

As mentioned earlier, Agrawal et al. [18, 19] paved the way for new research in association rule mining, especially in the context of market basket analysis. They introduced the Apriori algorithm for association rule mining. This algorithm has had high influence on the data mining community and it will be beneficial if an introduction is made here.

Like most of the association rule mining algorithms developed later, Apriori is composed of two phases. In the first phase frequent itemsets are found through repeated passes on the database. In the second phase these frequent itemsets are used in generating association rules. More details on Apriori algorithm steps are given next.

The first step in the Apriori algorithm is finding frequent 1-itemsets. The database

is scanned and a count list for each item is populated. This list is pruned according to the minimum support value so that only frequent items are left. The next step is the join step where a candidate list is generated from the previous frequent itemset list. Frequent itemset list is joined with itself, that is itemsets are grown by taking other itemsets into consideration. The candidate itemsets have one more item than the frequent itemsets that they are generated from. In the next step database is scanned again to find the support counts of each candidate itemsets. Again, these are pruned according to the minimum support value, and the final list is taken as the next frequent itemset. We start the cycle again by joining this frequent itemset list with itself to generate the next candidate itemset. The cycle stops when the new candidate list is empty. After the cycle has been broken we are left with frequent itemsets. Finding association rules from frequent itemsets is a straightforward job, where subsets of the frequent itemsets are taken and confidence values are checked with the minimum confidence value. Rules satisfying this threshold value constitute the results.

Apriori algorithm is said to be a bottom-up, breadth first search algorithm because of the way frequent itemsets are generated. Algorithm name is a reference to *a priori* knowledge used by the algorithm: Any subset of a frequent itemset must itself be frequent. This principle, also called anti-monotonicity or downward-closure property, is vital to the Apriori algorithm because it prunes the search space dramatically, thus enabling a big speedup.

Although widely known and implemented, Apriori algorithm suffers from many inefficiencies, and this has culminated in huge research into finding better algorithms. In the next section different frequent itemset mining algorithms will be briefly discussed.

This algorithm is presented as pseudocode in Figure 2.1 for clarity.

2.4 Frequent Itemset Mining

The astute reader would have noticed that finding frequent itemsets is especially important in association rule mining. This is due to the fact that finding frequent itemsets takes considerably more time than generating association rules from this data. Note how trivial it is to simply divide frequent itemsets as antecedent and consequent and compute the confidence values. As such, substantial research on association rule mining has focused itself on frequent itemset mining only.

Survey type of publications provide great information in summary format about ad-

```
L_1 ← ∅
I ← all the items occurring in all the transactions
scan database and find counts for all items i ∈ I
for item i ∈ I do
    if count(i) ≥ minsup then
        L_1 ← L_1 ∪ {i}
    end if
end for
k ← 2
C_k ← join(L_{k−1})
while C_k ≠ ∅ do
    scan database and find counts for all itemsets c ∈ C_k
    for candidate itemset c in C_k do
        if count(c) ≥ minsup then
            L_k ← L_k ∪ {c}
        end if
    end for
    k ← k + 1
    C_k ← join(L_{k−1})
end while
take the subsets of found frequent itemsets to generate rules
```

Figure 2.1: Apriori Algorithm Pseudocode

vances in a field. In writing this section I have made use of several survey papers and the introduction sections of many papers where a short history is almost always provided. Please refer to [44], [28] and [38] for three really well written survey papers summarizing many of the important developments.

2.4.1 Serial FIM Algorithms

Apriori algorithm had dominated the association rule mining scene from its inception in the early 1990s up to the early 2000s. During this time, many of the newly developed algorithms, including parallel ones, were derived from the Apriori algorithm [57, 58, 65, 74,

64, 27]. After this period, newer data structures and methods started to be investigated.

Apriori had several inherent inefficiencies: it would need to pass through the database a lot of times, and it generated a large number of candidate itemsets that would prove to be time consuming while checking the support values. One reaction to this problem was the FP-growth algorithm [46], which boasted not using a candidate generation method.

FP-growth algorithm came with a novel data structure called frequent-pattern tree (FP-tree) which would prove to be influential in the coming years. FP-tree is basically a prefix tree with some modifications. It uses prefix paths, which are tree nodes shared by different leaves so that the database is condensed. It is claimed that for dense databases where most transactions have shared items, FP-tree can compress the database on the orders of thousands of magnitude. Compression of this scale enables suitable databases to be wholly read into the memory, saving in I/O work needed. FP-growth algorithm makes use of FP-tree not only for database compression but also for finding frequent itemsets as well. A pattern growth approach, as opposed to candidate generation approach, is used in finding patterns. FP-growth algorithm has been extended or modified extensively in the literature [49, 33, 40, 60]. By now, extended tree structures had become widely used in database representation.

Up until now, all the algorithms we have looked at used the horizontal data format. In this most widely used format, the database is represented as transactions which are sets of items. This format can be seen in Table 2.1 depicted earlier. There is however, another data format that has been used by some of the algorithms. This is called the vertical data format. In this data format, each item has corresponding transaction IDs in a set. Thus, one can easily find all the transactions an item is existent in, without scanning the database. An advantage of vertical data format is this easiness of finding the support counts, simply counting the number of transaction IDs is enough.

An important algorithm making use of vertical data format is Eclat, proposed along with several other algorithms by Zaki [80]. Eclat makes use of the easy support counting vertical data format provides. It uses intersection operations to find support values of itemsets containing more than one item. By taking intersections between different transaction ID sets belonging to different itemsets, one can find the transactions containing both itemsets. Finding support values is then a trivial counting problem. Eclat generates candidate itemsets through this intersection method and the algorithm stops when the candidate set is empty.

In 1999, Pasquier et al. proposed a new format for representing frequent itemsets [59],

which is called the *frequent closed itemsets*. Previously, the whole set of frequent itemsets were generated along with the each subset. For example, the frequent itemsets result may have included both the rule *{milk, butter : 10%}*, which represents a 10% support value for milk and butter together, and the rule *{milk : 10%}*, which represents the item "milk" with the same 10% support value. Notice that the superset and the subset both have the same support and the subset is redundant as it can be generated from the superset. An important property to remember here is that a support value of an itemset can never increase as newer items are added. If $\{i_1\}$ is a 1-itemset with a support value of $x\%$, than an itemset which has one or more items added, $\{i_1, i_2, \cdots\}$ can only have support value of x% or lower, but never a higher support value. Frequent closed itemsets only include subsets if and only if the support value is higher than the superset. If the support value is the same (and it cannot be lower for a subset), than the subset is not included in the result as it can be generated from the superset and the support value can be copied exactly.

Closed itemsets became the center of attention and the default result format for many algorithms because of their ability to compress the results. One very popular example given by Han et al. [45] is described now. Take a database with two transactions of the form:

$$[\{i_1, i_1, \cdots, i_{100}\}, \{i_1, i_2, \cdots, i_{50}\}]$$

In this database, when the minimum support value is taken as one, number of frequent itemsets generated is an overwhelming $2^{100} - 1$ (all the subsets of the whole items set minus the empty set). Number of frequent closed itemsets on the other hand, is a mere 2, $\{i_1, i_2, \cdots, i_{50}\}$ with support value of 2 and $\{i_1, i_2, \cdots, i_{100}\}$ with a support value of 1. One can easily see that frequent itemsets generation can be a huge burden because of the sheer number of itemsets while frequent closed itemsets can easily compress the result set. The compression provided by closed itemsets is not the only reason for its adoption. Previously, *frequent maximal itemset* was also proposed [25] for compressed representation. Closed itemsets have the important property that they provide a lossless, complete compression. One can easily generate all of the $2^{100} - 1$ frequent itemsets from the two frequent closed itemsets if a need arises.

Many of the algorithms developed later used closed itemsets as the result format [61, 77, 81, 50], including the CLOSET+ algorithm, which is the algorithm used for parallelization in this thesis. CLOSET+ algorithm built upon many of the previous advancements made in the frequent itemset mining field. It makes use of the FP-tree structure to condense

the database and several methods previously developed to prune the search space. Many novel methods have also been proposed with this algorithm, including a newer method in pruning the search space and an FP-tree like structure called result tree which is used for checking closedness. The parallel algorithm described in this thesis makes heavy use of CLOSET+, and the mining part of the algorithm is executed on each process. For this reason, detailed information about the CLOSET+ algorithm is left to Section 4.1, with the parallel algorithm following in the same chapter. The section illustrating the parallel algorithm include examples similar to the ones used in describing the sequential algorithm, so as to make a comparison easier. In the next section, parallel data mining algorithms are elaborated upon.

2.4.2 Parallel FIM Algorithms

Just a year after the publication of the Apriori algorithm [19], a parallel association rule mining algorithm called PDM was published [58], itself based on a revised Apriori algorithm called DHP [57]. A year later, in 1996, Agrawal et al. published a research [17] detailing three parallelized versions of the Apriori algorithm. As was the case with serial algorithms, most of the parallel association rule mining algorithms developed up to early 2000s were also based on the Apriori algorithm.

In [17], Agrawal et al. present parallel versions of the Apriori algorithm. The new parallel algorithms described in the paper have Single Program Multiple Data (SPMD) form where the same algorithm (program or process) is employed by different processes and results are shared. In the newly presented algorithms, Apriori is the basic algorithm employed by different processes. Different data is sent or received by each process and the results are merged.

SPMD is claimed to be most used technique in designing parallel algorithms [32]. It is also used widely in parallel data mining. A presentation of different parallelization forms in the specific area of knowledge discovery can be seen in Talia's paper "Parallelism in Knowledge Discovery Techniques" [71].

Zaki et al. proposed another set of parallel algorithms in [79]. They made use of both vertical data format and maximal frequent itemsets. The parallel algorithms are similar to the Eclat algorithm discussed in the previous section, where vertical data format had been used for faster support counting. Data clustering is used so that the communication overhead is minimum between processes, which is one of the major causes of performance penalty in parallel algorithms. In addition, efficient lattice traversal methods had been de-

veloped for faster mining of patterns. Zaki et al. provide parallel algorithms for generating both maximal frequent itemsets and all frequent itemsets.

Han et al. proposed newer parallel algorithms [43] based on the Apriori and aimed to eliminate some of the inefficiencies existent. In [78], Zaïane et al. present MLFPT, a parallel frequent itemset mining algorithm based on the FP-growth algorithm described in the previous section. A more recent paper published in 2004 [48], describe the implementation of a similar parallel algorithm based on FP-growth that is implemented using a Message Passing Interface library.

FP-tree based parallelization approaches has become a hot topic partly because of the many advantages FP-trees bring. There has been a number of recent research focusing on better division of these trees and enabling fair loads for distributed architectures. As will be described in Chapter 3, most of the parallelization research has focused on multi-core shared memory architectures partly due to their wide use. Some of the more recent research include [51, 29].

Parallel frequent closed itemset mining is a relatively new subject. A recent paper published in 2007 [53] claims to be *"the first frequent closed itemset mining parallel algorithm proposed so far"*. This status of being relatively less researched made frequent closed itemset mining a good research area to work on. Coupled with the fact that closed itemset mining brings many advantages to data mining, it can be predicted that closed frequent itemset mining will be more researched in the near future, including parallel versions.

It seems beneficial to talk about the previous parallel frequent closed itemset mining algorithm in more detail here. The exact algorithm that has been parallelized in [53] is called DCI_CLOSED which makes use of vertical bitmaps for data representation and not tree structures as we have used for CLOSET+. The research as described in the aforementioned paper also emphasizes cache friendliness and SIMD extensions for the parallelization. By cache friendliness, it is meant that data is stored as vertical bitmaps and as a result these are small in size which means they can hopefully fit into the CPU cache. Another advantage of vertical bitmaps is that they reside in consecutive memory locations and this reduces cache misses. SIMD extensions on the other hand refer to newer CPU instruction sets which can be utilized for bitmap operations. In particular, DCI_CLOSED utilizes operations on bitmaps heavily, for example as intersection or inclusion operations. These operations are said to be much more efficient when native CPU instructions are used, one example of which is $POPCNT$ instruction which can count the number of bits set in a 64 bit word. Job division is described in the paper to be tested with both static and

dynamic division and the tests have been run on SMP architectures using up to 8 dual core CPUs with only one core utilized, ie. 8 threads have been utilized. Reported test results provide good speedups for various support values and data sets. Although it has not been elaborated upon, OpenMP is thought to have been used for thread communication for this algorithm implementation.

2.5 Current Problems and Research Directions

Data mining community faces numerous challenges. An abundance of research into the still popular areas are addressing many of the problems. For some of the problems, the problem content has changed over time, for others, the basic questions remain the same.

Performance considerations have been the most prevalent problem in data mining. This problem is compounded by the fact that data mining has many applications where the data to be "mined" is huge and the computations required are too much. Performance expectations have also increased over time and this has resulted in newer methods increasing data mining efficiency. Lowering processor and memory usage have become an aim for newer algorithms. Concurrency is also becoming important as a way in creating more responsive data mining applications. As multicore processors and distributed computing becomes more widespread, there is a pressure on newer data mining algorithms to use this huge computing power.

Another important issue in data mining is the more recent notion of *interestingness*. It is believed that association rule mining discovers too many patterns, including many of which constitute *common knowledge*, or information that is too evident and known at large. This is generally undesirable. Newer research in interestingness aims to discover information that is most likely to be useful and relevant [72, 36]. Interestingness is a delicate subject because it is hard to find objective measures fitting a situation. On the other hand, it might be hard to define what constitutes interesting, and this definition might be different for each user or each state of a configuration.

In association rule mining field, there have been newer research areas formed thanks to previous research. Maximal and closed itemset mining were both proposed to increase the efficiency of data mining, and they have themselves become topics of newer research. After the acceptance of merits of maximal or closed itemset presentations, newer algorithms were developed to further advance their mining. This is one example where a redefinition of a problem makes path for other research.

A recent research area is incremental mining (see [30] for an example). Incremental mining is a specialized data mining topic tailored for fast changing data. In some application areas, especially on the Internet, data gets updated very fast where minor changes occur frequently. As such, a need for faster model generation and easier update with the addition of new data has emerged. Incremental mining aims to create and update models with less work as new data comes in.

Privacy is another topic which is hot in data mining. With the rising popularity of social networking websites like myspace, facebook and others, mining data without compromising users' privacy has become even more important. A number of research addresses privacy preserving data mining techniques [20, 16, 76, 75]. Creating models and extracting patterns without the use of a user's individual data has been the aim of these methods.

Sometimes, newer application areas direct research. A nice example of this can be seen in colossal pattern mining [83] area. Colossal patterns are defined as large patterns where there are huge number of frequent patterns existing. These patterns could be found in specialized scientific research areas such as bioinformatics. There seems to be no silver bullet for solving each type of data mining task. Specialized algorithms tailored for data found in a specific field are developed with newer research. This concludes future directions section of this chapter.

Chapter 3

OVERVIEW OF PARALLEL PROGRAMMING

Parallel programming refers to numerous programming methods aiming to take advantage of parallel computing resources provided by computers. Parallel programming is based on the simple idea that having more than one program execution simultaneously for the same problem yields faster results.

Parallel programming is not a new concept as it has been used in various areas for many years, most notably in high performance computing. Despite this, parallel programming methods have seen a surge of interest in the recent years. This popularity is largely attributed to the rise of multi-core CPU technology and the popularity of cheaper cluster computing environments made totally of commodity hardware (desktop PCs).

Moore's Law (first introduced in [54], revised later) is defined as the observation and prediction that number of transistors (or components, as used in the original paper) in an integrated circuit doubles approximately every two years. This has resulted in exponential growth in computer performance. Traditionally, processor speed-ups in line with the Moore's Law had been mostly due to frequency gains by packing more and more transistors. While the Moore's Law is still valid, processor development has taken a new spin in the last few years. The old technique of increasing CPU frequency has recently been challenged because of various issues such as physical constraints (including the problem of packing up smaller microelectronics and heat dissipation problems), increased power consumption and the problem that memory lags behind processing power and becomes the bottleneck (the memory wall problem). The new trend in increasing processor performance is using the multi-core processor model wherein several processing cores are utilized in a single processor. Multi-core CPU model is claimed to address many deficiencies ex-

istent in the old model, and thus it is becoming widely used. Consumer processors now include quad-core CPUs with 4 processing units, while 8 core CPUs are becoming cheaper.

Another trend adding to the interest in parallel programming has been the use of commodity hardware in constructing cluster environments. Institutes and organizations are increasingly deploying high performance computing systems by making use of cheap commodity hardware. This has enabled growing deployment of computing clusters, especially at institutions which could not afford custom high-end servers and also broadened their use at those which could. At the peak of this trend seems to be Google, where thousands of computers are used in a wide range of jobs. It is claimed that computer clusters composed of commodity hardware "achieves superior performance at a fraction of the cost of a system built from fewer, but more expensive, high-end servers" [23].

How did the multi-core CPU development and the growing use of cluster computers helped parallel programming? To take advantage of the multi-core processor technology and shared nothing cluster architectures to enable faster running programs, one needs to make use of parallel programming methodologies. Several methods exist in achieving parallel programming, each with varying success and properties. Some of the more popular methods include parallel languages, automatic code generation techniques, implicitly parallel executing operating systems, POSIX Threads, message passing libraries and APIs for flexible thread management using compiler directives.

Before moving on to discuss parallel programming methods, definition of two important terms useful in the measurement of parallel programming will now be given.

Definition 3.1
Speedup of a parallel program presents how much faster it is than the corresponding serial version. It is defined by the following formula:

$$S = \frac{Execution\ time\ of\ the\ serial\ version}{Execution\ time\ of\ the\ parallel\ version}$$

Speedup is useful for determining how fast the new implementation is, but it is also used to find efficiency, which also takes the number of processors into account. Efficiency measures how well the processors added for computations are used.

Definition 3.2
Efficiency is used in measuring the utilization of added computational power by using the speedup and the number of processors added. More specifically, it divides speedup by the number of processors used. It is defined by the following formula:

$$E = \frac{Speedup\ shown\ by\ the\ program}{Number\ of\ processors}$$

3.1 Automatic Parallelization and Parallel Languages

In this section, programming language and operating system facet of parallelization will be discussed. Parallel languages, a much longed but little achieved subject, is the first topic of discussion. Distributed operating systems, which share some of the misfortunes of parallel languages, is the following topic. This section will hopefully convey why automatic parallelization has not gained a wide audience.

Several implicitly parallel programming languages have been published in the literature [35, 55]. Implicitly parallel programming languages promise the advantages of parallel programming in the looks of conventional sequential programming. They make use of compilers or interpreters to automatically ensure parallel execution. The "implicit" keyword in the language definition offers easy parallelism without the need for any extra work on behalf of the programmer. Parallelism aware compiler or interpreter analyzes the source code to generate parallel running machine code automatically. These languages do not provide the flexibility often needed in parallel programming. Programmers using these languages have little control in the parallel execution of their code, thus explicit parallelizing constructs are in general not needed. This is also considered as an advantage of these languages, but coupled with non optimal code generation provided by the compiler/interpreter, it has limited the popularity of these languages.

Similar issues are existent in cluster operating systems providing resource sharing capabilities to programs [52]. MOSIX is one such system [22] which is widely used. It is an operating system for managing cluster computing resources. MOSIX originally had derivations from several different Unix distributions, but it was later ported for Linux as well. Many features have been added to the existent operating system kernels for cluster management needs. The software layer added by MOSIX addressed resource allocation and sharing problems associated with the cluster environment. Load balancing, load sharing,

process migration, kernel communication and remote execution are some of the capabilities added or fine tuned. The advantage of MOSIX comes mainly because of its ability to automatically run, migrate and load balance processes dynamically with no supervision required by the programmer. The developed program does not have to use any external libraries and it does not need special compiler/interpreters. Execution is controlled by the operating system, a program can be executed on a remote node and migrated according to the load. To the program, the OS interface is the same, similar to any SMP system. This simple view is named with a self explaining term: single system image [62]. Like parallel languages, distributed operating systems also limit the control programmer has. As a result of the growing usage and maturity of other parallel programming methods providing additional control (mainly OpenMP and MPI), popularity of distributed operating systems have declined in the last few years.

There is a reason for the apparent failure of widespread acceptance of automatic parallelization methods. Although automatic parallelization methods keep up to the promise of easy automated parallelization, it suffers in the efficiency and performance areas, which are the basic reasons for using parallel programming in the first place. This performance penalty is caused by the inherent difficulty in analyzing and parallelizing source code. How will the code execution behave, which loops will be taken for how many times, which variables will be shared at which times and what will be the size of the data resident in these variables? These are all valid problems that need to be taken into consideration while parallelizing programs, and they are hard to predict. Until better compilers and operating systems are designed which can better predict an execution of a program, the solution is to use the programmer's intelligence in parallel programming.

3.2 Threads

Threads are defined as the concurrent executions of different tasks of a given program, and they are one of the most widely used methods in parallel programming. They are related with the "process" notion, but they offer flexibility and efficiency compared with processes. Process and thread implementations may differ from an operating system to another or even within programming languages, but in this thesis both terms will be used with their typical properties.

There are several important differences between processes and threads. These differences can be summarized in Table 3.1. One important distinction between a process and

a thread is that a process has its own address space while a thread shares it with other threads from the same process. This results in easy access of variables by threads while processes have to go through the tedious and slow interprocess communication mechanisms provided by the OS. Thanks to the sharing of the address space and state, threads are usually associated with less overhead, and thus are more efficient than processes. Nevertheless, there is a trade-off. Sharing the address space introduces new types of bugs through deadlocks, race conditions and other interesting behaviour when concurrent access to the same data is desired. Using semaphore-like constructs, one can ensure the proper concurrent access of variables by threads. Processes have their own address space and are thus protected from these errors.

Table 3.1: Processes vs. Threads

Processes	**Threads**
Processes are independent units of execution	Threads are tied to a process
Processes have their own address space and state information	Threads share the same address space and state within the parent process
Processes interact through interprocess communication (IPC) methods provided by the OS (harder to do, with security considerations)	Threads can use more flexible methods, such as seamlessly reading memory locations since the address space is shared
Switching between processes is costly	Switching between threads is relatively cheaper

Threads are the most used method for achieving concurrent execution. They are generally provided in a variety of programming languages so that programs can work on different tasks at once, and still be able to run efficiently. One popular usage of threads is in the user interface design. For example, several different user interface elements (eg. windows or tabs) can be controlled by different threads as it is mainly done in Java. Threads in user interface programming provides increased responsiveness and it avoids the hang-up or freezing of a window while another one is busy.

A disadvantage of the threads model is that it is not portable. There are various APIs for different platforms if one wants to use threads in parallel programming. The most popular ones are POSIX threads (pthreads) for *nix operating systems and WinAPI threads for Microsoft Windows operating systems. Furthermore, it can be a challenging task if one does all thread management by himself, without using any libraries or tools tailored for this specific job. It could be complicated to devise how and when new threads will be created, when they will be deleted or slept and how the jobs will be distributed. There have been various libraries put forward with the intend of automating these jobs. In the next section, OpenMP will be introduced which tries to address some of the problems.

3.3 OpenMP

OpenMP is a specification, a set of terms and their definitions, which enables parallel programming in shared memory environments. It has implementations supporting C/C++ and Fortran, working on many architectures such as Unix and clones (Linux) and Microsoft operating systems. OpenMP does not fit the definition of library, parallel programming is achieved by using compiler directives (or pragma definitions, as it is also widely used).

OpenMP uses multithreading for running tasks concurrently. It uses the general concept of multithreading where a master thread forks other threads which all run concurrently. If the tasks are finished, a join operation is conducted so that all the threads merge and only the master thread remains, like in the beginning. If more parallel tasks are forthcoming, a new set of threads could be forked again and this execution cycle could continue.

The cycle of creating new threads, merging and starting all over again can be best seen in Figure 3.1 (image taken from [13]). In the image, the top part shows a serial execution of various tasks, and the bottom part shows how it can be executed in parallel. We have to note here that parallel tasks 1 to 3 seen in the image all contain independent subtasks as shown. Had there been a dependency, parallel execution of the subtasks would not have been carried out. In this case, where tasks are dependent, serial execution must be preferred.

As described earlier, OpenMP uses preprocessor directives to achieve parallel execution. Also called pragma definitions, these directives are ignored by the compiler if OpenMP extensions are not installed and the compiler does not support it. Thus, serial versions of the program can be easily built, without any modifications (there are some

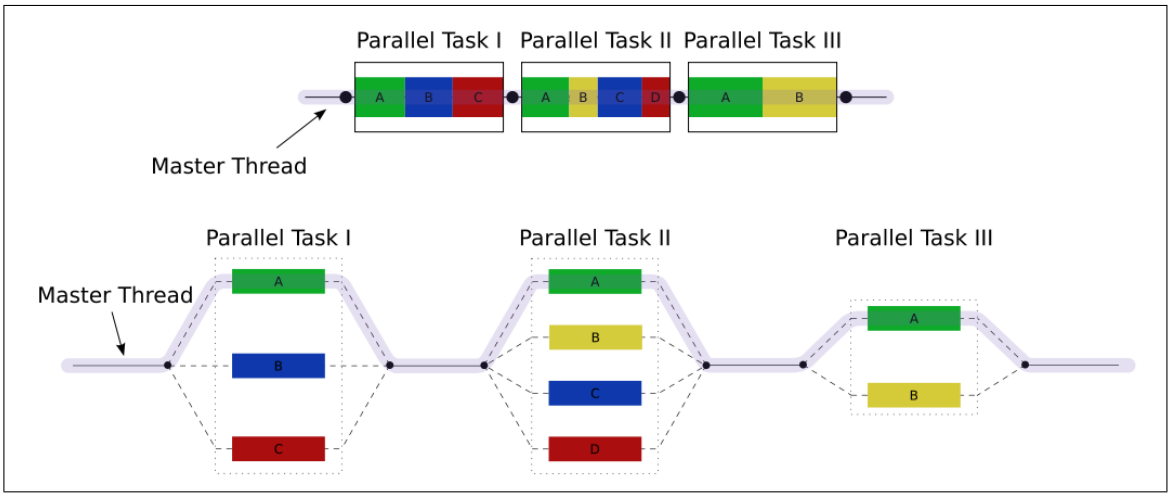


Figure 3.1: Execution model in OpenMP

caveats involved when more complex operations need to be done, and so serial versions need modifications, but advanced OpenMP is left to the reader).

A "hello world" program using OpenMP can be found in Figure 3.2. As can be seen, OpenMP pragma directives start with "#pragma omp". This simple example may give rise to the following question: How do we specify the number of threads we want? This and many other configuration parameters can be set using environment variables; thread number is specified by setting "OMP_NUM_THREADS" variable. This flexibility of setting numerous configuration options by environment variables is given as an advantage of OpenMP.

```
#include "omp.h"

int main(int argc, char* argv[])
{
  #pragma omp parallel
   printf("Hello, world.\n");
   return 0;
}
```

Figure 3.2: Hello World in OpenMP

It should be noted that while multiprogramming is easy and flexible with OpenMP, avoiding race conditions and deadlocks is left to the programmer. If tasks have dependence on each other, synchronization constructs such as barriers, critical sections and others can be used to avoid erroneous execution. Private variables can also be used within OpenMP to avoid races between threads accessing non-local variables and global data.

Table 3.2: Do-it-yourself Threads vs. OpenMP

DIY Threads	**OpenMP**
Threads need different libraries	OpenMP is implemented using compiler directives
Threads are not as portable, different libraries exist between different platforms (eg. Unix and Windows) and there are different libraries even in the same platform	OpenMP is portable, implementations conform to the standard
Parallelism has to be specified explicitly, high workload on the programmer	OpenMP takes care of most of the decomposition of work and data
Serial and parallel versions have many differences	Serial and parallel versions can be generated in general without any modifications (except for some advanced constructs where limited modifications are needed)

OpenMP is a widely popular solution to parallel programming. It provides various advantages over custom threading implementations. Scalability, simplicity and flexibility concerns are addressed to some extent. In the shared memory architectures, OpenMP promises satisfactory performance, and the latest specification OpenMP 3.0 released on May 2008 is expected to improve some of the shortcomings (eg. tasks are added to OpenMP) as more and more of the compiler producers embrace it.

OpenMP is basically a multithreading solution saving the programmer the hassle of maintaining threads and parallelizing a work. In Table 3.2 differences between a custom multithreaded implementation and OpenMP is summarized. OpenMP provides flexibility, scalability and simplicity on shared memory systems. In the next section we will be looking

at message passing techniques, which provide parallelism in shared nothing architectures. By using OpenMP and MPI in a hybrid application, one can parallelize a program to run on multicore distributed clusters.

3.4 MPI

MPI is a specification describing an API for communication between different computers (and hence the name message passing interface). It has been used widely in the high performance computing community, starting much before multicore processors were common, to program distributed memory architectures. MPI has proven itself to be the de facto standard for many applications using distributed memory, but it can be used for shared memory programming as well. It uses the general concept of message passing, of which one of the earlier examples had been PVM (Parallel Virtual Machine) [70]. PVM also happens to be one of the main motivations for a message passing standard.

MPI works with processes, as opposed to threads, and thus it directly inherits process properties. Refer to the earlier comparison in Table 3.1 for more information. MPI provides an API for communication between different program processes working on a problem. Normally each core is assigned a single process for maximum performance. MPI enables detailed configuration and more processes can be assigned at the cost of processor performance. If more than one process is assigned to a processor core, it is said to be oversubscribed and processes share the CPU time.

What happens when MPI is used in a shared memory environment? If a program can run in distributed memory architectures, then there should be no problem running in shared memory architectures as well. Even so, having a multithreaded approach would increase the performance of the application because threads are more suitable in shared memory systems. MPI standard does not require implementations to be multithreaded and different implementations have taken different approaches. Multithreading is not supported at large by implementations. Still, there is some room for improvements in process handling when shared memory architectures are used, such as direct memory copying. This is used by the Open MPI implementation.

Similar to the example source code given in the OpenMP section, MPI equivalent is presented in Figure 3.3. From the "hello world" example, it can be seen that MPI is used by calling library functions, as opposed to compiler pragma directives used in OpenMP. Although both source codes are given in C, MPI can be used with C++ and Fortran too,

as it is the case with OpenMP.

```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
  int myrank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Processor %d of %d says: Hello World!\n",
                              myrank, size);

  MPI_Finalize();
  return 0;
}
```

Figure 3.3: Hello World in MPI

MPI provides many functions for communication between processes. These include point-to-point communication constructs where two processes are connected or collective operations including more than two processes. MPI also provides ways to group processes according to any way programmer likes. Furthermore, virtual topologies can be chosen so that MPI can simplify and increase the efficiency of communication between processes. There are also many variations in communication functions according to the communication format. It can be synchronous where the functions block until the message is safely shared between the parties, or it can be asynchronous where processes continue their work and check later whether the operation has completed successfully. Collective communication also has many forms including synchronization constructs (eg. barriers), scatter, gather and reduction operations. These provide low level flexibility to the programmer, he is free to choose the best strategy to use.

MPI naturally supports the SPMD form of parallelism by spawning multiple copies of the same program and enabling communication in between them. Nevertheless, MPI implementations also allow different programs to be spawned as well, with communication between them intact. Thus, MPMD (Multiple Program Multiple Data) form can also be implemented easily.

You can see a comparison of OpenMP and MPI in Table 3.3. The most obvious difference to take into consideration is that OpenMP works on shared memory architectures while MPI works in both [24].

Table 3.3: OpenMP vs. MPI

OpenMP	**MPI**
OpenMP is implemented using compiler directives	MPI uses library function calls
OpenMP uses the thread model	MPI uses the process model
OpenMP is suited for shared memory programming	MPI is best suited for distributed memory programming, but shared memory programming can also be used
OpenMP has simple parallelization of loops and other tasks, demands less work on programmer's side	MPI has more flexibility and low level configuration options. There's more work for the programmer to do
Serial versions in OpenMP is often trivial	Serial version without the MPI overhead needs "ifdef" directives that clutter the code or an MPI stub library containing dummy code

Among the hottest topics in MPI development community include "fault tolerance". This is important in grid computing systems where the computing elements making up the architecture could be physically distant and system wise heterogeneous. Fault tolerance is resuming operation as a whole if a failure occurs in some part of the system. There is continued development in this area in many MPI implementations.

Chapter 4

IMPLEMENTATION

In this chapter, implementation information for the performed research is presented. The research is focused on the parallel implementation of a fast frequent itemset mining algorithm called CLOSET+ [77]. Accordingly, the first section is about the CLOSET+ algorithm in its serial form, as described in the original research paper by Wang et al. After this, the solution framework will be presented to better understand the implementation and the design decisions taken. Modifications enabling parallel execution will be presented afterwards, concluding this chapter.

4.1 CLOSET+ Algorithm

CLOSET+ algorithm is a fast algorithm using the widely popular FP-tree (Frequent Pattern tree) structure. It is composed of two phases. In the first phase, a compact representation of the database using FP-tree structure is built. After this phase, second phase commences where the FP-tree is mined and frequent patterns are found. CLOSET+ makes use of several novel techniques for pruning the search space and thus increasing the mining speed. More information about the FP-tree structure and building it in the first phase will be described in the next section while the second phase, covering actual mining will be described in the following.

4.1.1 Building the FP-tree

FP-tree structure was first proposed with the FP-growth algorithm [46] published by Han et al. It became a widely used format for representing the database to be mined thanks to its high compression ratio. Compression is vital to any mining algorithm because performance can be greatly enhanced if the whole database can be compressed to fit into

memory and thus no time is wasted on the costly I/O operations from the disk. FP-tree is an extension of prefix tree structure, where a node in a tree can represent an item existing in more than one database transaction.

FP-tree structure may be best explained by an example. The following example is taken directly from the CLOSET+ paper [77] to facilitate the use of the original paper as reference, should the reader need it. This example will also be useful in the coming sections, since it will be used directly with the parallel version to enable an easy comparison.

Assume the database contents are as in Table 4.1. The first thing to do is finding the support counts of each item and pruning those items having less than minimum support. Assume that the minimum support value is two. When we sort the support list, it becomes $f\_list = (f : 4,\ c : 4,\ a : 3,\ b : 3,\ m : 3,\ p : 3)$. Notice that the list is called f_list and that *d, g, i* and *n* has been pruned. Items having less than the minimum support value are not frequent and they are pruned. Remember the downward-closure property from the Apriori Algorithm section, it is straight forward that every frequent itemset must itself contain frequent items only and because of this rule, infrequent items can safely be discarded.

Table 4.1: An Example Database

TID	Basket contents
001	a, c, f, m, p
002	a, c, d, f, m, p
003	a, b, c, f, g, m
004	b, f, i
005	b, c, n, p

The next step in building the FP-tree is sorting the database according to the *f_list*, while discarding infrequent items. In Table 4.2 this can be seen.

The final step in building FP-trees is inserting each sorted transaction to the tree one by one. In particular, each transaction starts at the root of the tree. The first item is searched in the children of the root, if it is found, that node's count is incremented. If the item is not one of the children of the root node, then a node with that item and count of

Table 4.2: Pruned and Ordered Database

TID	Pruned & ordered items
001	f, c, a, m, p
002	f, c, a, m, p
003	f, c, a, b, m
004	f, b
005	c, b, p

1 is added as a child. The second item will start from this newly inserted or incremented node and the operations will be similar. This will go on until all items included in the transaction have been finished, in which case next transaction will start at the root of the tree, just like in the beginning.

In Figure 4.1 one can see the shape of the tree after each processing of the transactions. The numbers below the trees show the ID of the last transaction that has been processed and inserted. For example, the first tree represents the shape of the tree after the first transaction has been processed. As can be seen from the figure, new nodes are inserted with a count of 1 while existing nodes' counts are incremented. When all the transactions have been processed, whole of the database has been compressed and represented in a tree, this is the fifth tree in Figure 4.1.

As the shared items in transactions in a database increases, items belonging to more transactions are represented by the same nodes in the tree, and this increases the compression rate. This compression is not clearly evident in our simple example described above. Databases where many items are included in most of the transactions are called *dense* databases. It is claimed that FP-tree structures can compress databases in the order of hundreds or thousands of magnitudes for dense datasets [77].

4.1.2 Mining the FP-tree

After the construction of the FP-tree, the database will not be read again. All the mining work will be done by processing the FP-tree. CLOSET+ mines the trees by projecting each node to newer FP-trees and employing three pruning techniques called "item merging", "sub-itemset pruning" and "item skipping".

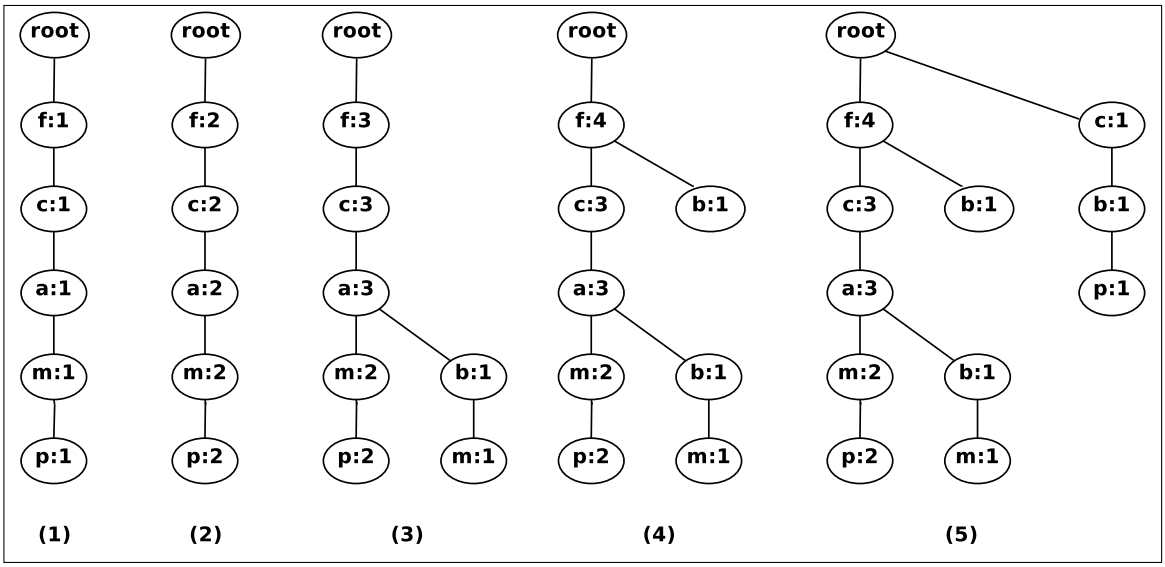


Figure 4.1: Building of FP-tree as each transaction is processed

Item merging technique prunes the search space by merging two itemsets if they are always found together. In other words, if itemset Y contains all the items existing in every transaction containing the itemset X, then $X \cup Y$ is taken as a frequent closed itemset. There is also no need to search for closed itemsets containing X and a subset of Y as they will be represented by $X \cup Y$.

Sub-itemset pruning technique on the other hand does not mine unpromising trees if they are descending from itemsets which are proper subsets of already found itemsets with the same support value. This technique asserts that if itemset Y is a proper subset of an already found frequent closed itemset X with the same support value, then Y and all the itemsets descending from Y in the tree do not need to be taken into consideration as they will be represented by the frequent closed itemset X.

In addition to these methods, a newer method similar to the two pruning techniques described is the item skipping technique which lowers the number of recursive tree projections. By the use of this method, if an item has the same support value in a projected tree, it can be pruned from the parent tree. That is, the same item does not have to be mined in the parent of the currently projected tree. This technique also speeds up mining because it eliminates the redundant work.

All three of the methods aim to prune the search space the algorithm has to work on, and thus to increase its efficiency. In the remainder of this section other aspects of the algorithm will be analyzed.

In the original research, two methods for mining the trees are presented, according to the nature of the dataset. For dense datasets, a bottom-up physical tree projection is proposed while for sparse datasets a top-down pseudo tree projection is proposed. However, no metrics are presented for determining which method is the best for a given database. Determining if the database at hand is dense or sparse and whether it is more efficient to use a top-down or a bottom-up projection is completely left to the reader. Since FP-trees are better suited for dense datasets, and since both methods are similar to each other in spirit, only the bottom-up physical projection method will be discussed in this section.

The one structure missing in the FP-tree examples presented in the previous section is the links between the nodes and the header table providing a simple index. FP-trees incorporated links between nodes to enable easier traversal between nodes having the same item and thus increased mining speed. The header table on the other hand allowed for the access of the nodes in a bottom up manner as needed. These additions which had not been included in Figure 4.1 can be seen in Figure 4.2.

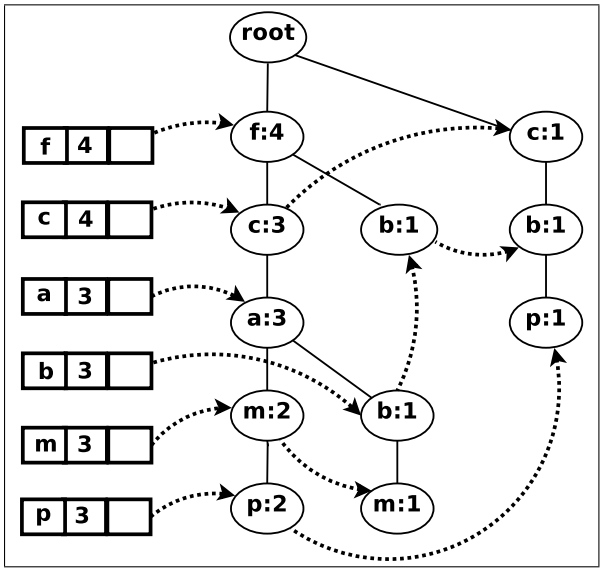


Figure 4.2: FP-tree with side links

In the actual mining of the FP-tree structures, projected trees are constructed in a bottom up manner using the header table. These projected trees are then either projected themselves or frequent itemsets are added to the result tree by the use of the techniques discussed in this section. A simple example mining step will be now presented.

In the first step, a projected tree for the bottommost item in the header vector will be constructed. For this example projection, conditional database for item p:3 is generated from the global FP-tree. Conditional database of an item is composed of transactions which include that item. In Figure 4.2 it can be seen that transactions including item p are both $\{fcam : 2\}$ and $\{cb : 1\}$. Conditional databases include the support values for these transactions, these support values also apply to each item existing in the transaction. Projected FP-trees are constructed in the same way as the global FP-trees are constructed. Conditional database is processed to find the support values of items in the transactions, f_list is constructed, transactions are pruned and sorted according to the f_list and inserted to the new tree. In Figure 4.3 projected tree for item p:3 can be seen.

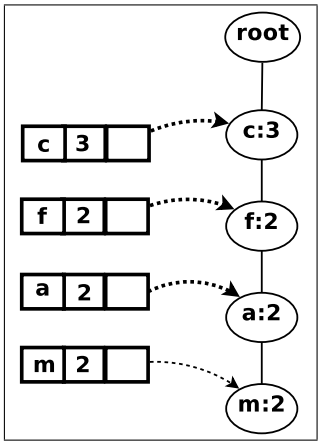


Figure 4.3: Projected FP-tree for item p:3

After constructing the projected FP-tree, mining starts with a similar bottom-up approach. The bottommost item, which is m, is added to the prefix itemset and a new conditional database is generated. The new prefix is pm:2 and the only transaction in the database is $\{cfa : 2\}$. By using the item merging technique, prefix and the conditional database is merged and $\{pmcfa : 2\}$ is inserted to the result tree as the first frequent itemset.

Going one node up, item a is the next item to be added to the prefix. Here, another pruning technique comes to the stage as the prefix $\{pa : 2\}$ is a proper subset of the first frequent itemset found. Sub-itemset pruning is used here and the mining can continue with upper nodes. The next item, f also shares a similar fate as the new prefix $\{pf : 2\}$

is also a proper subset and it is also pruned.

We move to the last item which is found in the top node, item c. Prefix for this node becomes $\{pc : 3\}$, thanks to the higher support value of the item. The conditional database for this node is empty as there are no nodes in the higher levels, and so the prefix can be inserted to the result tree directly. Itemset $\{pc : 3\}$ becomes another frequent itemset found.

After all the projections are mined for a node, upper nodes in the global FP-tree are mined. The same bottom-up traversal is used and tree projections for m, b, a, c and f are created in that order. Tree projections and the mining steps are similar to the first step described above, and will not be described here.

As candidate frequent closed itemsets are found, how are they checked to be really closed? This is done through a two level hash index pointing to nodes in an FP-tree like structure. A modified FP-tree structure is used in compressing the results, much the same way as compression achieved for the database. The nodes in this result tree are referenced by a hash index and this provides a way for checking if there exists a subset or a superset of the newly found itemset with the same support value. Thanks to the result tree structure, and related two level hash index, only closed frequent itemsets are stored.

Please refer to the original article [77] for a through discussion of the CLOSET+ algorithm.

4.2 Solution Framework

In this section, important design issues are elaborated upon regarding the programming framework chosen for the project. Some of the important questions center around the choice of MPI and its implementation, programming language, auxiliary libraries that were used and how debugging was done. These make up the rest of this section, it is hoped that each question is addressed satisfactorily.

4.2.1 Programming Language

During the decades of research into high performance computing, the two languages that have come to dominate this area are C (and C++, if one thinks of it mostly as a superset of C) and Fortran. As a result, these two languages have the most support when it comes to diverse parallel programming methods. On the other hand, high level dynamic languages such as Python and Ruby are receiving growing acceptance and the programming

enjoyment they bring with newer methodologies is apparent.

CLOSET+ and our parallel modification uses trees and graph structures which would highly benefit from the objected oriented approach high level programming languages provide. For the implementation of our algorithm, many MPI libraries for different languages were investigated. There exists two different methods for supporting MPI-like message passing in different programming languages. In the first method, a native library implements part of the message passing API interface with slight modifications in the function names and sometimes inner workings. For the second method, a wrapper library is developed that uses an MPI library under the hood. Both of these methods had been checked for different libraries and it was found that there is some loss of flexibility with all these libraries. Some features defined in the MPI standard has not been implemented or not supported completely in these languages.

To take advantage of the object oriented programming methodology, C++ was chosen as the programming language. It has extensions to C for object oriented programming and MPI libraries written in C can be used for low level work by simply calling them from the C++ program. To have a consistent interface and clean programming style, Boost libraries were chosen to interface MPI. Boost provided object oriented programming interface to the MPI library. More information about this library will be given in the next section where auxiliary libraries are discussed.

4.2.2 Libraries

Several high level open source libraries were used in our project for different purposes. This enabled the reuse of proven source code developed and tested by many people in the open source community.

The first library used in the project is the tclap library [14] which facilitated the easier parsing of command line options and pretty printing of the help contents. As our project focused on measuring speeds of different executions, many command line arguments were needed. In addition, because there were different types of data that could be processed, data type was also parsed from the arguments. This resulted in a rather long program execution command, which was easily programmed using tclap.

The second and the most important library used in the project is the Boost libraries [3]. Boost libraries are peer reviewed set of more than 80 libraries aimed to complement the missing parts of the Standard Template Library in C++. It has high quality libraries designed for many tasks, and some of the libraries have been accepted for incorpora-

tion into the C++ standard. Boost also has an MPI interface designed to enable object oriented usage of the MPI interface. Many function calls become object methods using encapsulation and an easier interface is offered for the many MPI functions. In Figure 4.4, a comparison is given where both raw MPI and Boost library version is shown for the functionally equivalent code segment.

```
// C Style MPI Function Calls
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0)
...

// C++ Style MPI Programming Using Boost Libraries
mpi::environment env(argc, argv);
mpi::communicator world;
if (world.rank() == 0)
...
```

Figure 4.4: MPI Programming Comparison

Boost MPI library does not only provide C++ style MPI interface, it also provides methods for easy transfer of complex data types. One disadvantage of MPI has been the complexity involved when a variable of type other than the primitive data types (integer, character, float etc.) is needed to be shared between the processes. This task involves MPI_Pack and MPI_Unpack methods and the custom serialization of variables. It is extremely tedious and error prone for the programmer to write this code by hand. Sending even simple string variables is not a trivial task in C++, while sending more complex data types such as trees and graphs can be extremely difficult.

In our algorithm, there is a need to send and receive graph-like structures with added constructs such as header tables linking to nodes in the graph or multi level hash indexes. Sending these structures correctly is an extremely complex job when one thinks about the edges, information contained in nodes such as variables and the links helper structures contain. Links should be followed and newer structures should be serialized as found, but

there is also the need to keep track of the pointers so as to not send the same structure more than once should it be linked from more than one place.

This complexity is the topic of a recent paper published in April 2008 by Tansey et al. [73]. The said paper compares different methods providing relief for the programmer who wants to send non-primitive data. Among various methods analyzed, Boost libraries are found to one of the best solutions to this problem. Tansey et al. also describe a tool they have developed which generates automatic serialization code, but as of January 2009, this tool has not been made available to the public.

By using both the Serialization and the MPI libraries provided by Boost, one can easily send non-primitive structures through MPI. The programmer does not have to deal with the low level intricacies of serialization and deserialization, pointer tracking and other complicated tasks. Most of the STL provided containers such as maps, vectors, lists and standard library provided string data type can be sent and received automatically. Custom data types such as graphs can be sent and received by making small additions to the class code. By implementing a simple serialization interface on a class, objects belonging to that class can be communicated between different processes.

Underlying MPI library used by the Boost MPI interface is an important topic as it is central to the communication between different computing nodes. For this reason, message passing deserves a section of its own and it is presented next.

4.2.3 Message Passing

A serial algorithm is generally parallelized with the hope of making it faster and more efficient. As described in the *parallel programming methods* chapter, MPI has been the *de facto* standard for applications using distributed memory. MPI gives the programmer ability to programme distributed memory architectures whereas OpenMP provides support for shared memory architectures. Coupled with the fact that MPI can also be used for shared memory programming, this flexibility gives MPI an edge over OpenMP. With the modern MPI implementations copying memory directly from one process to another in multi-core architectures, the choice of using MPI becomes natural.

After the selection of MPI, the exact implementation to be used appears as the next issue. The two major implementations of MPI are Open MPI [12] and mpich, which are both supported at the major grid centers in Turkey. Both the cluster at the Computer Engineering Department (CENG) and the National HPC Center (UYBHM) have these two libraries installed. After an analysis, it was found that both of these libraries are

exceptionally well and similar in spirit. Nevertheless, Open MPI was chosen because of its commitment to a better implementation, its flexibility and low level configuration options.

Open MPI is an MPI library representing the merger of several MPI libraries. It was envisaged that by taking the better parts of each library and bringing the good ideas and technology from different implementations, the best MPI implementation could be developed. Open MPI provides many low level configuration options such as using different network interconnectivity options, memory checking support, direct memory copying on shared memory architectures, optimizations for many options including when running on homogeneous machines etc. One interesting feature of Open MPI is that it can use different network connectivity options by itself, whereas for mpich different implementations are needed to connect on different networks. For example, mpich library working over InfiniBand is called mvapich.

Open MPI uses TCP for setting up different processes and management of non-user level MPI work. For shared memory architectures, the default user level data transfer method is direct copying of memory from one process to another. This relieves the processes of communication overhead and increases the connectivity speed. It should be noted that this will still be inherently slower than OpenMP, where threads are able to directly access the memory locations. Processes in MPI have their own address space and thus they are more flexible than OpenMP threads but this incurs some performance penalty. On the other hand, MPI processes have less danger of race conditions and locking of variables (via mutexes in threads) because they do not share their memory address space with other processes.

4.2.4 Debugging

Debugging is an important issue in parallel programming, as it is also the case in serial programming. Debugging even serial programs can be at times tricky, while parallel programming brings new types of bugs and complexity on the table. With the parallel execution, race conditions and deadlocks can occur, causing erratic behaviour or the freezing of the programs. Number of interleavings for a given MPI program can be huge, and it can be impossible for the human brain to think about each of these schedules. For example, an MPI program with 5 processes each making 5 MPI calls has an overwhelming number of execution options (much more than 10 Billion!). Assuming that there are no synchronization calls, that each MPI call is independent of each other, the exact number

becomes:

$$\frac{25!}{(5!)^5}$$

The sheer hugeness of the number of interleavings for a parallel program calls for good tools to aid in debugging. During the course of the project STLFilt [2], vim [8], valgrind [68], gdb [69] and ddd [82] has been used for debugging [5]. STLFilt filters C++ STL Template errors and shows them in a way programmer can better understand. The vim is an all round great text editor, but with its quickfix feature it can speed up the notorious edit-compile-debug cycles static compiled languages impose. Valgrind on the other hand is a suite of tools very useful for profiling and finding memory leaks. Using valgrind with Open MPI requires Open MPI libraries to be compiled with memory checking options set, see the Open MPI user guide for more detailed information.

The main tools used for debugging are the gnu project debugger gdb and its front-end graphical user interface, ddd. To be able to use these debuggers, one must compile the program source code giving the *-g* flag to the compiler so that the debugging information is included. Another important detail is the optimizations issue. To experience a smooth debugging session, optimizations must be turned off. Flags beginning with -O, such as *-O3* must be removed from compiler options. This is needed to ensure that the source code and the executable is consistent with each other. If optimizations are turned on, the compiler may choose to remove variables, change the way loops work and alter the code in other ways, which can confuse the debugger and cause irregular behaviour or even worse, a segmentation fault.

Gdb and ddd are two popular programs used widely for serial program debugging, but they can also be used for parallel debugging. There exists debuggers which have been developed for parallel use specifically, such as the popular TotalView, but they are commercial software and cost a lot of money in terms of software licences. Commercial debugger TotalView is not installed on the CENG HPC Cluster, but it is on the UYBHM cluster site. Natively parallel debuggers are actually not needed as serial debuggers can be used for parallel debugging just as well.

There exists three methods for debugging parallel programs using Open MPI. In the first method, *mpirun* command runs the debugger which in turn starts the application. Subsequently, a window is opened for each process. If gdb is preferred to be used, it should be started with a terminal emulation program, such as xterm, since gdb does not have a window of its own. If a graphical front-end like ddd is preferred, it can be run directly. In

Figure 4.5, one can see how both debuggers can be used.

```
$ mpirun -np 2 xterm -e gdb ./myProg
$ mpirun -np 2 ddd ./myProg
```

Figure 4.5: Running an Open MPI program through a debugger

After starting the debuggers, one can use standard debugging commands, insert breakpoints, run the program, inspect variables and so on.

The second method in debugging parallel programs is running the programs freely and attaching to the relevant process ID as needed. To use this method, a global debug function can be declared printing its process ID and sleeping until an attachment is made by a debugger. After attaching to a process, one can use debugger methods to break out of the sleep loop by changing the value of the control constant. This method of attaching to processes has some advantages compared to the other methods because one can make use of run time controls and selectively attach to processes as needed. For example, a conditional statement such as an *if* can be used to call the *debug* function when the process rank is some predetermined value, or if the process is in a predetermined state. An example function used for debugging by this method is given in Figure 4.6.

```
void Globals::debug() {
  int i = 0;
  char hostname[256];

  gethostname(hostname, sizeof(hostname));
  printf("PID %d on %s ready for attach\n", getpid(), hostname);
  while (0 == i)
      sleep(5);
}
```

Figure 4.6: Code segment used for attaching to an Open MPI program

Globals is a static class whose methods can be called from anywhere in the program without the need for creating an object. Typically the programmer determines a possible location of error and calls this function near that place in order to attach to the program by a debugger. In addition to this, one can call this function at the beginning to attach and debug a program right from its beginnings. Debugger programs have options for attaching to running programs by accepting the process ID. After the attachment, the loop control variable i's value can be changed via the debugger and conventional debugger commands could be used.

It should be noted that for the previous methods to work on a remote server, X forwarding must be supported by the remote server. This has security implications and thus many system administrators do not enable this feature. Debugging on a remote cluster has another side effect, processors are blocked until the whole debugging session finishes, which can take a long time. For these reasons, it is recommended that parallel application debugging is performed on the local computer.

The last method for debugging does not need X forwarding. This method makes use of MPMD form of parallelization and it is restrictive on the number of processes that can be debugged. Open MPI supports the concurrent execution of multiple programs and provides communication just as it is provided to concurrent execution of a single program. Using this strategy and keeping in mind that the command line is attached to the first running process, one can debug the first process easily. The command for using this method is given in Figure 4.7.

```
$ mpirun -np 1 gdb ./myProg : -np 1 ./myProg <arguments>
```

Figure 4.7: Debugging an Open MPI program using the MPMD strategy

As can be seen, each process uses the same executable, but the first one is run using the debugger. The debugger is in control of the first program while the second one is free in its execution. Like in the previous cases, one can use debugger options in debugging the first process.

This concludes the methods for debugging parallel applications using Open MPI. It now seems beneficial to describe how debugging can be done with other MPI libraries such as mpich. Different MPI libraries have implemented different levels of support for debugging.

The method used by mpich (or its InfiniBand version, mvapich, for that matter) is easier and more flexible than what Open MPI provides. In mpich, one can provide *-gdb* argument to mpirun and get access to gdb command prompts on all processes. Inputs to each gdb instance is tunneled by mpich and the debugger outputs are printed on the same screen. Multiple copies of the same message are discarded from each debugger and thus only one instance of a message is shown. It is possible to send debugging commands to a subset of the gdb instances. Similar to the Open MPI method, one can also attach to a running MPI process using mpirun. In Figure 4.8 one can see the commands for both options.

```
$ mpirun -gdb -n 10 ./myProg
$ mpirun -gdba <jobid>
```

Figure 4.8: Debugging an mpich program

Job ID's of running processes can be obtained from the mpich process manager, using the *mpdlistjobs* command.

4.3 Parallel Implementation

In this section, the actual implementation of the parallel CLOSET+ algorithm will be discussed step by step from the beginning to the end. The parallel algorithm, with its major steps is shown in Figure 4.9. The first three steps in the figure have been investigated in the literature previously. These steps involve the generation of FP-trees, a very popular structure in frequent itemset mining research. For one of the earliest research utilizing FP-trees for a parallel algorithm, please see the paper by Zaïane et al [78]. The remaining two steps shown in the figure has not been proposed in the past literature to the best of our knowledge, and they comprise of the division of FP-trees for mining and the merging of FP-tree like result trees. This division of mining on FP-trees and the merging of the result trees is the novel part of our algorithm, where the mining task is borrowed from the serial CLOSET+ algorithm. A similar research to Zaïane's work using MPI for parallelization of a different algorithm making use of FP-trees can be found in [48].

1. The data is divided at the transaction level. Each process is assigned a local database.
2. Each process calculates item counts from their local database. Local item counts are merged from each process to generate global item counts. Global item counts are distributed to all processes.
3. Local FP-trees are created on each process, again using the local database and the global counts. Local FP-trees are merged from each process. The global FP-tree is distributed to each process.
4. Mining task is divided using the header table of the global FP-tree. Each process mines its share of the global FP-tree using the serial CLOSET+ principles.
5. Local result trees are generated on each process, and these are merged to create the global result tree.

Figure 4.9: Parallel algorithm steps

4.3.1 Parallel Algorithm Steps

In this section each step is described in detail. More specific MPI and implementation details will be left for the following sections.

In the first step, the data is divided at the transaction level. In our implementation, data is divided so that each process is responsible for the processing of roughly the same number of transactions. This number is calculated simply by dividing the number of transactions to the number of processes. Each process then goes through that number of transactions reserved for itself, this data is called the local database. The last process is also responsible for the extra transactions if the number of transactions is not an integer multiple of the number of processes.

Example database division configurations will now be described with respect to the previous database depicted in Table 4.1. This example database has 5 transactions. If it is to be divided for 2 processes, the first process will be responsible for transactions 1 and 2, while the second process will be responsible for the rest of transactions, which is the transactions with IDs 3, 4 and 5. If the number of processes is 5, each process will get the same number of transactions to process, namely, a single transaction.

It should be noted that this division does not entail a perfectly fair workload, as transactions may include widely fluctuating number of items. Even so, this simple method provides a fair enough work division, it is estimated that taking the number of items and their nature in a transaction into account for load balancing would result in a big computational overhead. Assuming that each transaction has roughly the same number of items and that each process roughly gets the same number of unique items in their local database, it becomes clear that the efficiency is increased. Trade-offs of this sort where extra computation is needed for a perfectly fair workload are encountered frequently for the algorithm, in almost all cases it is decided that perfectly fair workload is not a wise decision in terms of performance.

In the second step each process goes through their local database and calculates item counts. This calculation is performed similar to the serial CLOSET+ algorithm. Local item counts data found by each process is than merged to generate the global item counts. Merging of local item counts is a trivial task, an example merging is shown in Figure 4.10. In the figure, local counts from two processes are shown merged to global item counts. The data used for this example is from the earlier database given in Table 4.1. Since we assume there exists two processes, transactions 1 and 2 are used by the first process to generate the first table in Figure 4.10. The second process on the other hand uses the remaining three transactions and generates the second table shown in the figure. The "+" sign in between the first two tables denotes the merging operation, which results in the third table. As seen in the figure, the merge operation simply adds the local counts together. Once the item counts for the whole data is found through merging, it is distributed to each process. In the end, each process has knowledge about the item counts regarding not only their local data but the whole database.

With the global item counts data at hand, each process can create its FP-tree by processing the local data. At the beginning of the third step, a pruning operation is performed on each transaction the process is responsible for. In this pruning task, items having less support value than the specified minimum is removed from the transaction. The remaining items in the transaction are than sorted with decreasing support values. In constructing FP-trees, each sorted and pruned transaction is inserted to the tree. This FP-tree construction is as it is described in the original article first proposing this structure [46].

After the local FP-trees are constructed at each process, they are merged to create the FP-tree representing the whole database. The merging operation on the FP-tree structures

Item	a	b	c	d	f	g	i	m	n	p
Support	2	0	2	1	2	0	0	2	0	2

+

Item	a	b	c	d	f	g	i	m	n	p
Support	1	3	2	0	2	1	1	1	1	1

=

Item	a	b	c	d	f	g	i	m	n	p
Support	3	3	4	1	4	1	1	3	1	3

Figure 4.10: Merging of the local item counts

is conducted using a method based on its semantics. In merging the trees, a node in one tree is taken and inserted to the other tree if it is not already present. The count of the node is also preserved in this case. If the node is already present in the tree, the count values are simply added. This node insertion and update operation is performed along with the traversal of the tree in a depth first way, from the root to the leaves. Merging of FP-trees have been implemented in previous research [78].

An example merging of two trees is seen in Figure 4.11. The referenced figure follows directly from the previous examples. The first two trees in the figure are generated using the same database division as described in the previous two steps. Minimum support value is taken to be 2, and so some of the items having smaller support in the merged table shown in Figure 4.10 is pruned from the database. These pruned items are not shown in the local FP-trees. After the merging of local FP-trees, the global FP-tree representing the whole database is distributed to each process. Local databases are no longer needed, memory used by the individual transactions and the local FP-tree can be freed since the global FP-tree will be used during mining in the subsequent steps.

An important detail to note here is the replication of FP-tree in different processes. The global FP-tree has the needed important parts of the transaction database, where by important parts we mean data which is going to have an effect on the list of frequent patterns. Only those items that are frequent are taken into consideration and items with support values less than the minimum are discarded, they are not existent in the FP-tree. This pruning, along with the compression FP-tree structures provide compact data

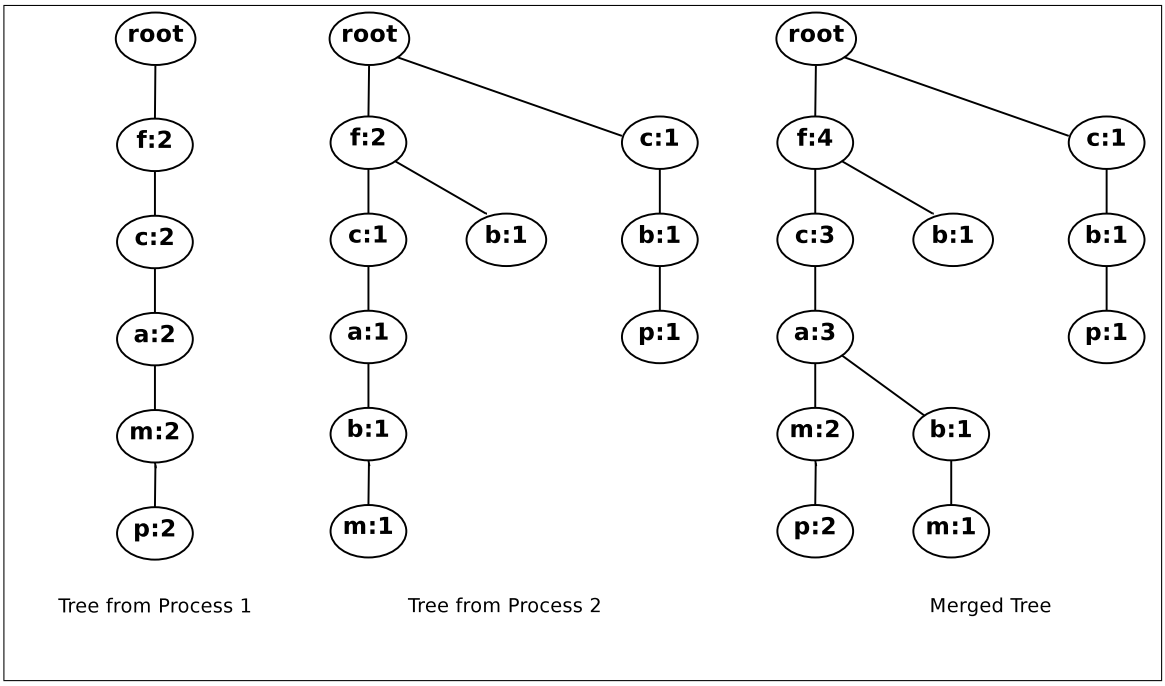


Figure 4.11: Merging of two FP-trees

representation especially on dense datasets. As a result, replication of FP-trees is not a vital problem.

In the fourth step, a work division is made and the actual mining task is performed on each process. The work division is done with the help of the FP-tree and its header table. Since the global FP-tree is available to all processes, they will all be working on the tree in Figure 4.2, including the header table and the side link pointers. The work division is made such that each process is responsible for mining equal number of items in the header table. During this division, the header table is simply divided into approximately equal number of rows and given to processes for mining. Bottom-up mining has advantages such as mining longer, potentially superset itemsets before shorter subsets. This is the reason continuous blocks from the header table are given to processes. The division is similar in spirit to the transaction level division in step one, and each process mines the header table entries in a top to bottom manner. The mining task itself is as it is described in the CLOSET+ paper [77]. Each process builds result trees, local trees built from the parts of the FP-tree they are responsible for.

For the last step of the algorithm, another tree merging operation is performed. This time local result trees from each process is merged to find the global result tree, which is the result tree containing all of the frequent closed itemsets for the whole database.

Merging result trees is inherently different from the previous FP-tree merging method because the semantics of the trees are different. Result trees represent closed frequent itemsets whereas FP-trees represented the transaction database. For this reason, merging method is different from how merging is performed with the FP-trees. Result trees are merged such that if a node is new, that is if it does not exist currently in the tree, it is inserted as is, with the support value intact. If the node already exists in the tree, than the support value is taken to be the higher one. This is in line with the closed itemsets concept.

Continuing with the same example used in the previous steps, Figure 4.12 shows the merging operation of two result trees. The first two result trees are from each processes, they are merged to form the third tree. Notice that nodes which do not exist in one of the trees is inserted directly whereas the larger support value is used for nodes existing in both of the trees. Similar to FP-tree merging, a depth first traversal is performed as each branch is processed.

This concludes all of the steps in the parallel algorithm. Merged result tree obtained in the last step contains the frequent closed itemsets for the whole database. In the next sections, more details about the implementation will be given.

4.3.2 Implementation Details

In this section some of the important implementation details will be unveiled. The questions which will be addressed include the creation of the serial version of the application, data types that are supported, and the exact MPI methods used for communication.

In the next chapter, program execution test results will be given. One of the executed tests compares the parallel program with the serial version. By serial version we mean a program having minimal MPI overhead, with no MPI overhead being the ideal. How was the serial application created? There are two methods used in the literature for creating the serial version of an MPI program. In the first method the parallel source is linked against a dummy MPI library that acts similar to the real library but does not do anything. It returns logical values for function calls, for example, the function call to getting the number of processes return 1, as you would expect. While returning logical results for MPI calls, the dummy library does not set up MPI communication layer and other tasks normally done, and so it does not have the overhead.

The second method in creating a serial version is a classic method. It involves surrounding the MPI code around preprocessor conditional directives. This method is selected for

the project. An excerpt of source code using this method is given in Figure 4.13.

In Figure 4.13, conditional directive "ifdef" is used to choose either using MPI calls for getting rank and world size data or not using any MPI calls but setting them with logical constant values. In generating the serial executable, conditional directives are used around MPI calls throughout the source code. Once this is done, one can define macros such as USE_MPI either by environment variables or by using the "-D" flag given as an argument to the compiler. By using a Makefile and defining rules using different macro definitions, one can easily generate various executable versions catering to different needs.

MPI provides various options for sharing data between processes. In the implementation of the parallel algorithm multiple options were sometimes possible. For some of these options, more than one method got implemented when the most efficient method was not apparent, for others a decision was chosen. In the case where multiple methods have been implemented, these can either be chosen through program arguments or through the use of different executables, generated through preprocessor directives as described previously.

There are three methods used with MPI for sharing data in the implementation. In the first method, all_reduce function is used and this is the easiest method as it takes care of the merging of data from each process and distributing it. The programmer has little burden when this method is used. In the second method, both reduce and broadcast functions are used together, effectively the data is first reduced at a single process and then distributed via the broadcast call. This method embodies two calls as opposed to the single call in the first method. In the last method, simple send and recv functions are used in sending and receiving the data. Due to the complexity of using this method when a large number of processes are existing, it is only implemented for two process architectures.

Another implementation detail to note is the different types of data that is supported for processing. The first data format supported in the project is the widely used retail database format where each line in a file represents a transaction and items in the transaction are separated by space characters. The second type of data supported is the type IBM Quest Synthetic Data Generator is able to output. This data type is useful as the data generator can generate made up data featuring various properties. The number of items to have, average number of items in a transaction, number of transactions are just some of the properties of data that could be controlled by the IBM data generator. In the next chapter, program tests conducted on both data types are shown.

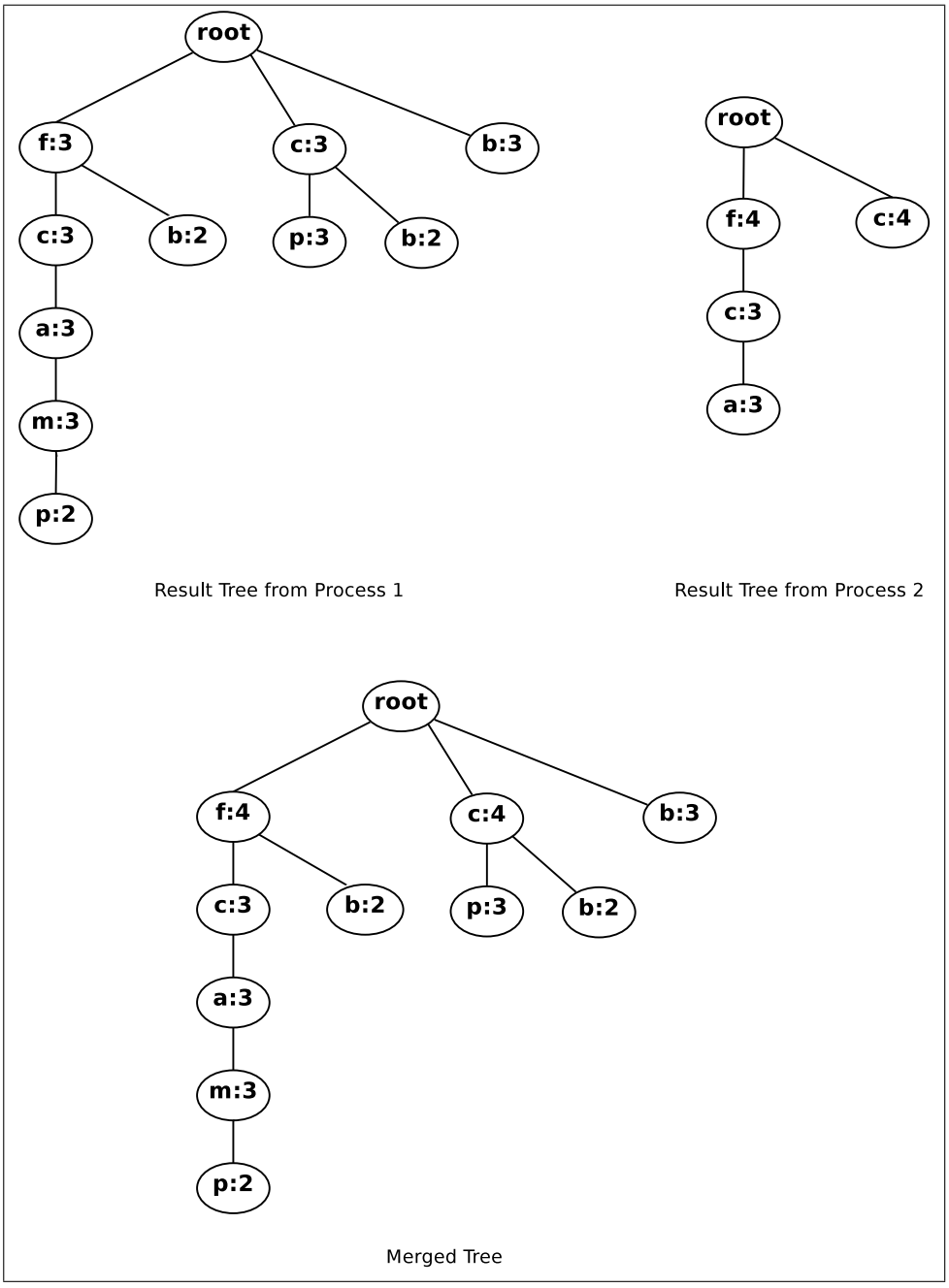


Figure 4.12: Merging of two result trees

```
int main(int argc, char* argv[]) {
    // To keep track of elapsed time.
    clock_t fp_start, sp_start;
    // Start the clock.
    fp_start = clock();
#ifdef USE_MPI
    /* Before initializing MPI, start a block so that the mpi is
       finalized at the end of the block and we can use the end
       time precisely. */
    {
    // Initialize MPI.
    boost::mpi::environment env(argc, argv);
    boost::mpi::communicator world;
    Globals::rank = world.rank(), Globals::size = world.size();
#else
    Globals::rank = 0, Globals::size = 1;
#endif
. . .
```

Figure 4.13: An excerpt from main.cpp

Chapter 5

EXPERIMENTAL RESULTS

This chapter presents various tests conducted with two different datasets showing different characteristics. Different aspects of parallel programming in MPI have been investigated and tests performed to verify predictions that had been made.

All the tests have been implemented using automated shell scripts and related graphics have been generated using the gnuplot application. A short discussion of the results follows each graphic, while a more encompassing commentary and conclusion is left to the ending chapter. Information about the computing environment tests were run on and the properties of the two datasets which were used in the tests are the first topics of discussion.

5.1 Computing Environment and Datasets

All the performance tests described in this chapter have been carried out on the cluster of the High Performance Computing laboratory at the Department of Computer Engineering of the Middle East Technical University. Hardware properties in each computing element found in this cluster can be summarized as:

- **Processor:** 2 x Intel Xeon E5430 Quad Core CPU 2.66 GHz, 12 MB L2 Cache, 1333 MHz FSB
- **Memory:** 16 GB
- **Connectivity:** 2 x 3Com 4200G 24 Port Gigabit Ethernet Switch & Voltaire 9240D 24 Port Infiniband Switch

In terms of software, the cluster has both Open MPI and mpich/mvapich libraries installed along with multiple compilers such as gcc (GNU Compiler Collection) and icc (Intel C++ Compiler). All the necessary programs from the GNU tool chain also exist on

the servers, should the programmer need them; and there is also the possibility to compile and install custom libraries in user's home directory if a need arises. This is how Boost libraries have been used; newest Boost library was compiled and installed in the user's home directory. The compiler used is the popular gcc with level 3 optimizations, allowing the compiler to use inline functions, variable tracking and many other methods taking advantage of full optimization.

Testing an implementation using various datasets is important in measuring the performance of data mining algorithms because many times a "one size fits all" approach does not work. In data mining field, successful algorithms generally work well for a specific type of data exhibiting desired properties. For many FP-tree based algorithms, a dense dataset is the desired data type. One reason for this is the fact that FP-tree structures provide high compression for dense data. This in turn enables the whole database to fit into memory while FP-tree also guiding the mining process. For this reason, many algorithms are fine tuned for dense datasets.

At this point a more elaborate discussion about dense and sparse datasets would be beneficial. There is no standard definition of denseness or sparseness that is found in the literature. Even in prior publications showing research that are specifically designed for one type of data, a clear test determining if a data is dense or sparse is not given. Nevertheless, it is widely accepted that dense datasets include more items for each transaction while sparse datasets have less number of items out of a large pool of potential items. In this respect, it is much more easier, and perhaps more correct, to say that a dataset is more dense or more sparse than another, rather than a more certain approach cutting clearly which data is dense and which is sparse.

For the testing process of our implementation two datasets were used; a synthetic dense dataset and a sparse dataset taken from real life data. The aim in using two datasets with different properties is to see the difference in performance gains with respect to different data. Information about these two datasets is outlined in Table 5.1, enabling an easy comparison.

The synthetic dense data has been generated using the IBM's Quest Synthetic Data Generator [9]. This application allows the programmer to control many aspects of the data. To generate a dense dataset, small number of items were used with high average transaction size.

The second dataset on the other hand contains real data from a retail store. It has been downloaded from the dataset repository at the University of Helsinki (FIMI). More

Table 5.1: Properties of the two datasets

Dataset	Number of Items	Number of Transactions	File Size
Quest	100	125155	180 MB
Retail	16470	88163	4 MB

information and an analysis on this data can be found in [26].

It should be noted that the performed tests utilized all the cores on each processor before moving on to the next processor; and all the processors on each computing node before moving on to the next node. More specifically, tests utilizing up to 4 cores have used only a single processor since the processors are quad core CPUs. Tests utilizing up to 8 cores run on the same computing node, using both of the processors since each computing node includes two processors. For executions utilizing more than 8 cores, other computing nodes are used, utilizing cores in the new computing node in a similar way. This scheduling mechanism is the default method in Open MPI, and it is called *scheduling by slot*. Another important remark is that all the processes in the tests have run on a dedicated processor core. In other words, oversubscription, where more than one process is run on a core, is not allowed. As a final note, the tests were run up to ten times each and the run times were averaged to make up for any anomalies an execution may result in. As a result, all the graphics showing timings use average run times.

5.2 Results

In this section, the actual test results will be presented. First graphics to be investigated is regarding the number of frequent closed itemsets each data set yields, with respect to the support value. Graphs presenting the relation between the number of frequent closed itemsets and the support value can be seen in Figure 5.1, for the both datasets investigated.

In the above figure, exponential growth of itemsets is evident with respect to the support value. As the support value decreases, the growth in the number of frequent closed itemsets increases very fast. Recall the distinction between closed and non-closed itemsets as described in the previous chapters; the actual number of frequent itemsets is much more than the frequent closed itemsets number depicted in the graphics. Nevertheless, number

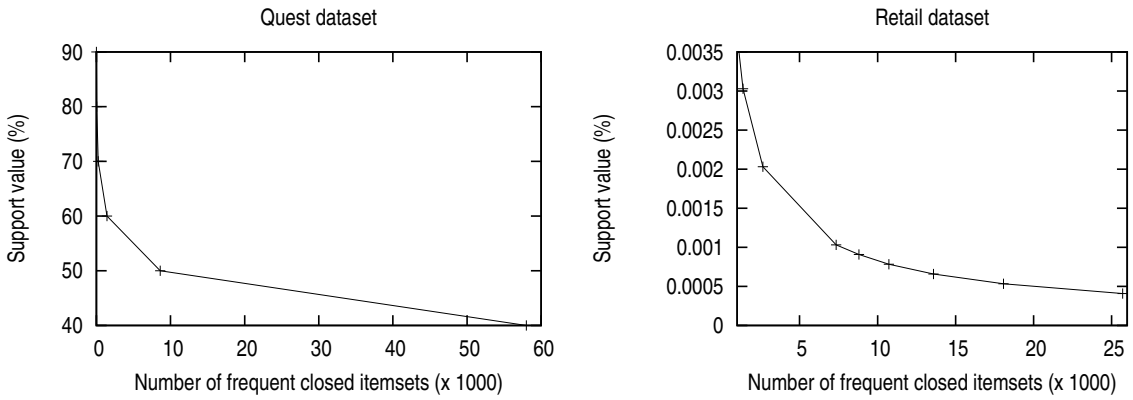


Figure 5.1: Relationship between support and frequent closed itemsets

of frequent closed itemsets can still be in the order of thousands as evident in the graphics. For the support value of 40%, Quest dataset provides nearly 60 thousand frequent closed itemsets. Retail dataset on the other hand, provides about 25 thousand frequent closed itemset for the support value of 0.0004%.

A dataset property which can be inferred from the figures is the denseness or sparseness characteristic of data. In the Quest dataset, number of frequent closed itemsets are very high even when the support value is as large as 40%. In the Retail dataset however, the support values need to be less than 0.005% to yield similar number of frequent closed itemsets. Even then, the number of frequent closed itemsets is much less than what is found for the Quest dataset.

The next set of graphics to be discussed in this chapter is about the run time comparison of the MPI application with respect to different number of processes. Keep in mind that during the testing of the implementation, each process was allowed to run on a dedicated processor core. This means that processors are not oversubscribed, and so the maximum performance is attained.

Next three figures present execution time with respect to the support value for runs utilizing different number of processor cores using the Quest dataset.

Figure 5.2(a) is restricted for runs up to 4 cores, so as not to overcrowd the graph. In the next figure, Figure 5.2(b), one can see the runs utilizing 4, 8, 12 and 16 cores. A similar figure depicting a more detailed view by plotting the executions up to a higher support value can be seen in Figure 5.2(c).

The first striking observation to be noticed from all the figures is that adding more processors decreases the execution time highly for a given range of support values, albeit at

a slower rate as more processors are added. In the tests, the highest value of efficiency (as defined in Chapter 3) has been observed for the execution with 2 cores. A comparison of speedup and efficiency values for runs utilizing up to 4 cores, using the Quest dataset with 90% support value can be seen in Table 5.2. As can be seen from this figure, the speedup values increase as more processor cores are added, whereas the efficiency values decrease. The reason for this phenomena is the communication costs induced by maintaining more processes. As each process executes the parallel CLOSET+ algorithm described in the previous chapters, they need to communicate partial results and synchronize at determined points. This communication puts a burden on the processes and the efficiency is decreased by this overhead. This is also the case when support values are lowered. The efficiency decreases as the support threshold is lowered, precisely because the communication costs are increased. As support threshold is lowered, larger partial results are generated which increases the overhead of communication. As a result, efficiency decreases.

Table 5.2: Speedup and efficiency comparison (support value 90%, Quest dataset)

	1 Core	2 Cores	3 Cores	4 Cores
Speedup	1	1.84	2.48	2.98
Efficiency	1	0.92	0.82	0.74

Similar graphics for the second dataset, the Retail dataset, can be seen in the four figures beginning with Figure 5.3(a). In the first figure, execution times for runs utilizing up to 4 cores can be seen. Since this is a sparse dataset, the support threshold is reduced to get higher number of itemsets and execution times. A more detailed view of this graph, depicting execution times for support values lower than 5% can be seen in Figure 5.3(b). Similar to the previous graphs for the Quest dataset, runs utilizing more than 4 cores can be seen in Figure 5.3(c) with a more detailed view in Figure 5.3(d).

As evidenced by the figures, runs utilizing more processors begin to get behind as the support threshold is lowered. In Figure 5.3(d), it can be seen that the critical support value where it is still beneficiary to use more processor cores is 2%. After this point, the run utilizing 4 cores begins to perform faster than other runs utilizing 8, 12 and 16 cores. This is in line with our earlier comment regarding the communication overhead when more

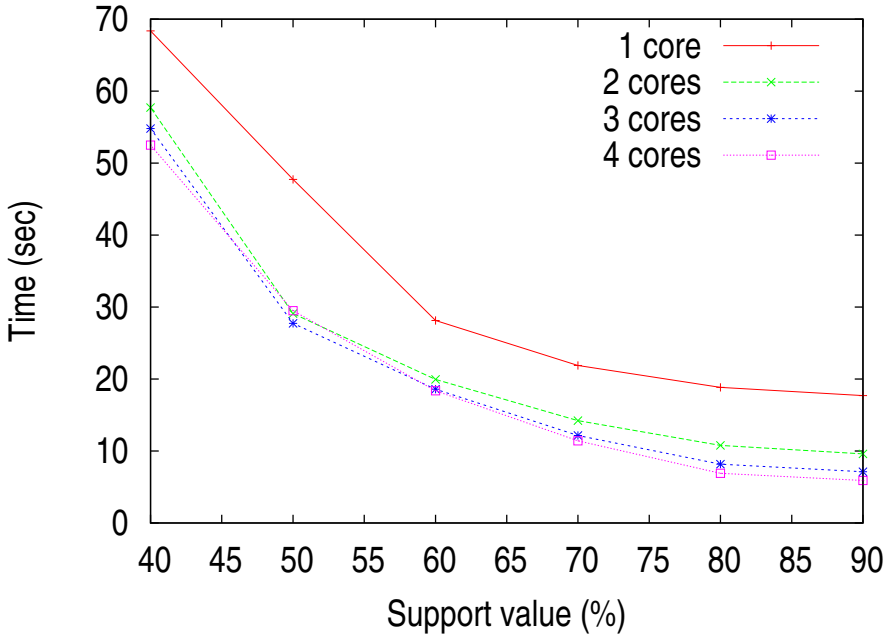
cores are added or when the support threshold is lowered.

By observing the graphs, one can see where the execution times are heading when more than 16 cores are to be utilized. As can be seen from the graphs with 1-4 core and 4-16 core executions, timing gains becomes lower as more processing units are added, lowering efficiency. The run times get lower for higher support values but as support threshold is lowered, it begins to get higher than executions with less processing cores. There is a trade off between communication costs and computational gains obtained by utilizing more processing units. The other parameter in this trade off is the support threshold, which effects the size of the data to be communicated. As in other trade offs, the optimum value of the number of processors depends on various factors including the support threshold and the data type (its denseness or sparseness for example) that is used in the real working environment.

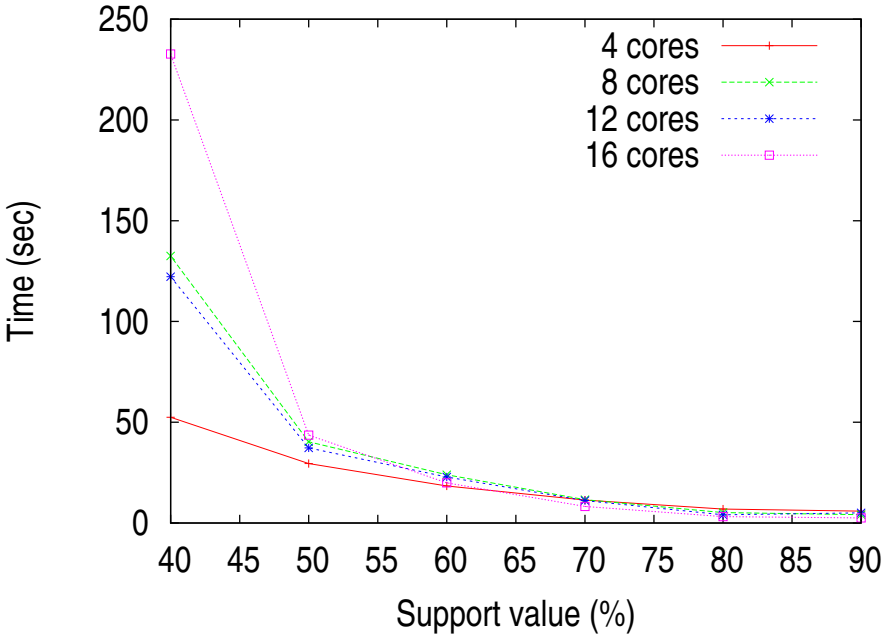
The parallel CLOSET+ algorithm can be seen to be more efficient for denser datasets from the figures showing execution times. This observation is similar to the serial algorithm. In the serial case, dense datasets enable compressed FP-trees to be much more smaller enabling a faster processing. FP-trees are basically modified prefix trees, and they can be easily seen to become much more smaller in size as the data they represent becomes more dense. In the parallel case, there is a similar advantage, but there is also an advantage in communication costs. This is because of the fact that prefix tree structures are communicated between computing nodes as partial results are shared. In MPI applications, communication overhead is a limiting factor and compressing the communicated data to make it easier to send and receive could cut this overhead and enable a more efficient execution.

Up until now, all the executions utilizing a single core still included the MPI library linked to the source. These runs still contained MPI calls, and so execution control was left to MPI to decide. To test the completely serial version of the implementation, compiler preprocessor conditionals have been used to remove parts of the code including MPI calls, as described in Section 4.3.2. A comparison of executions of completely serial program and the other executable containing MPI calls can be seen in three figures beginning with Figure 5.4(a). These graphics show execution times of the serial executable and the MPI version run with a single process. In Figure 5.4(a) and Figure 5.4(b), a comparison of execution times can be seen for Quest and Retail datasets respectively. A more detailed view of Figure 5.4(b) can be seen in Figure 5.4(c), showing part of the graph where the execution uses lower support values.

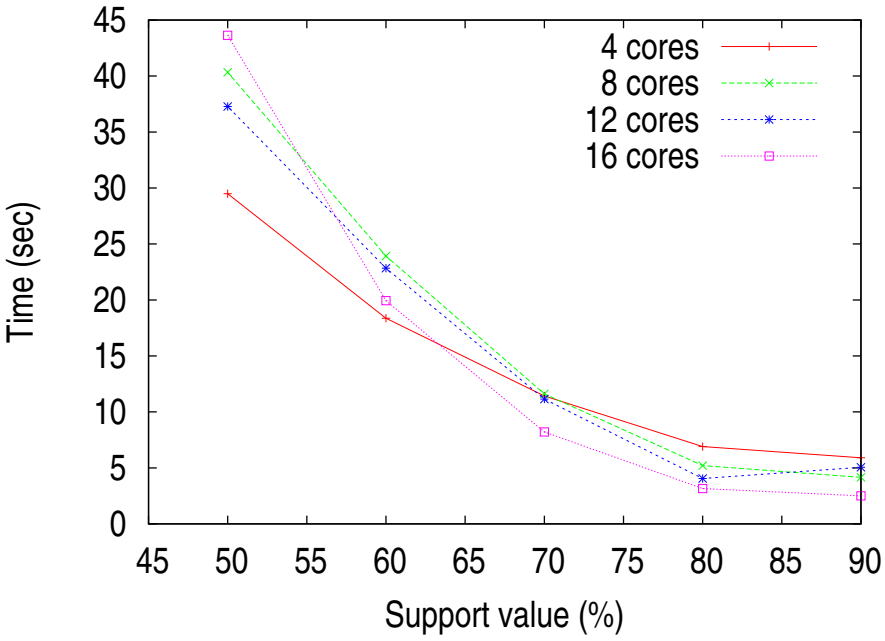
The basic conclusion obtained from the graphs is that the MPI overhead is mostly negligible for 1 process executions. Even so, in performance critical applications, a stripped off versions containing no MPI calls would be beneficial. The classic method using preprocessor directives is well known, easy to maintain and suitable for this task.



(a) Running on 1-4 cores

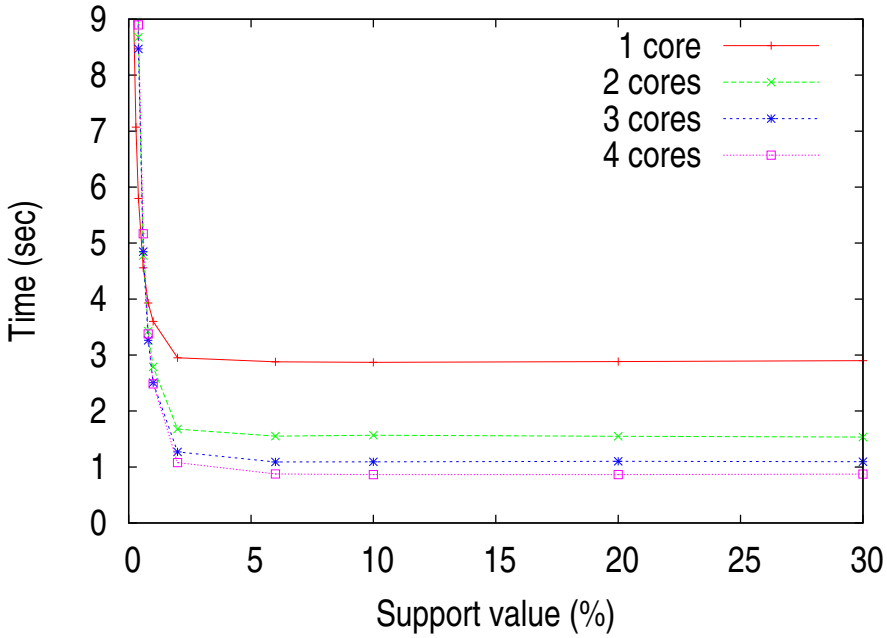


(b) Running on 4, 8, 12 and 16 cores

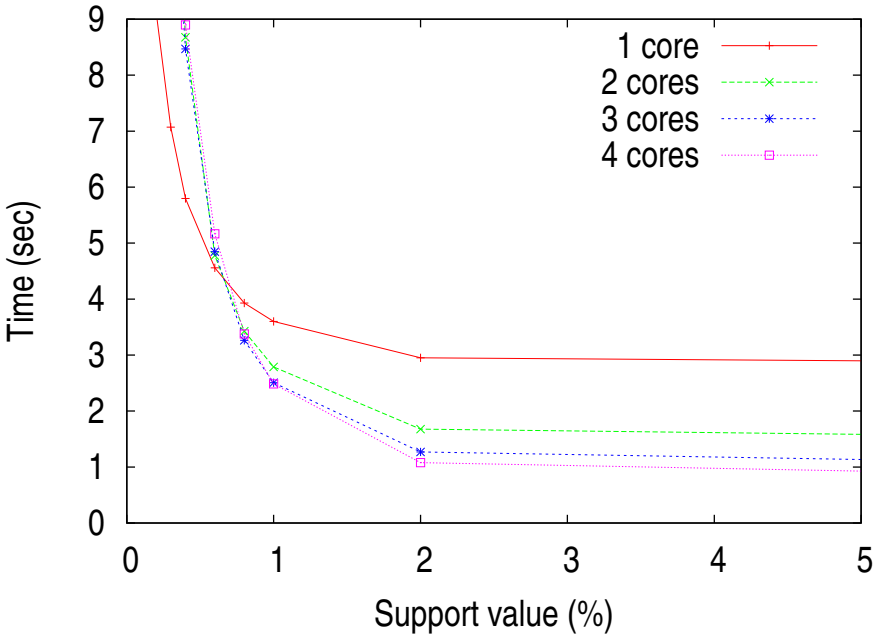


(c) Running on 4, 8, 12 and 16 cores, detailed

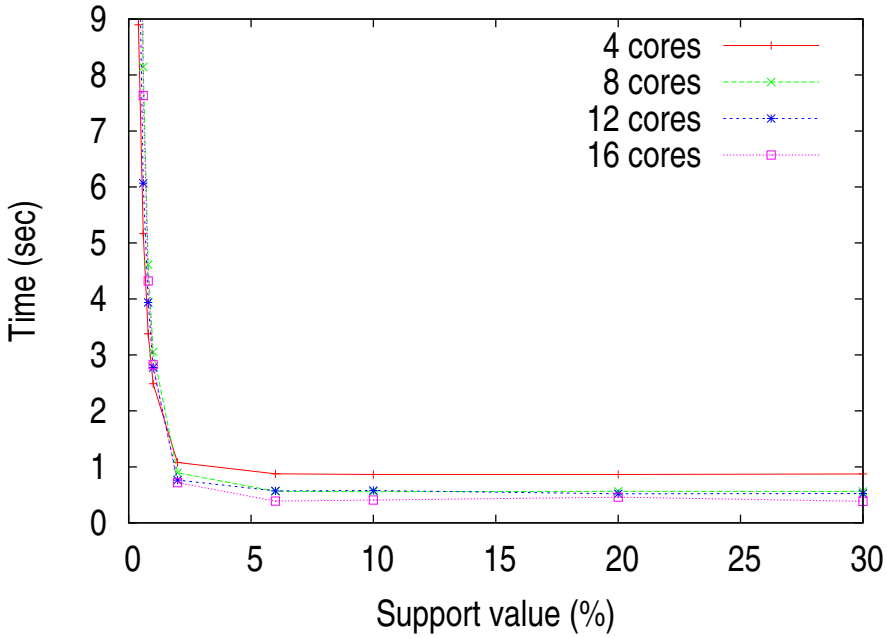
Figure 5.2: Execution times using the Quest dataset



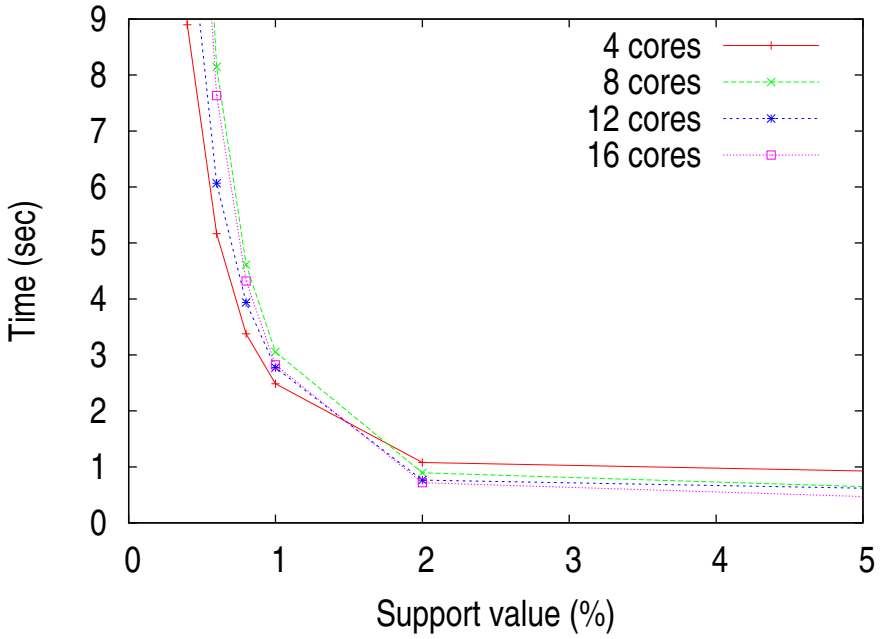
(a) Running on 1-4 cores



(b) Running on 1-4 cores, detailed

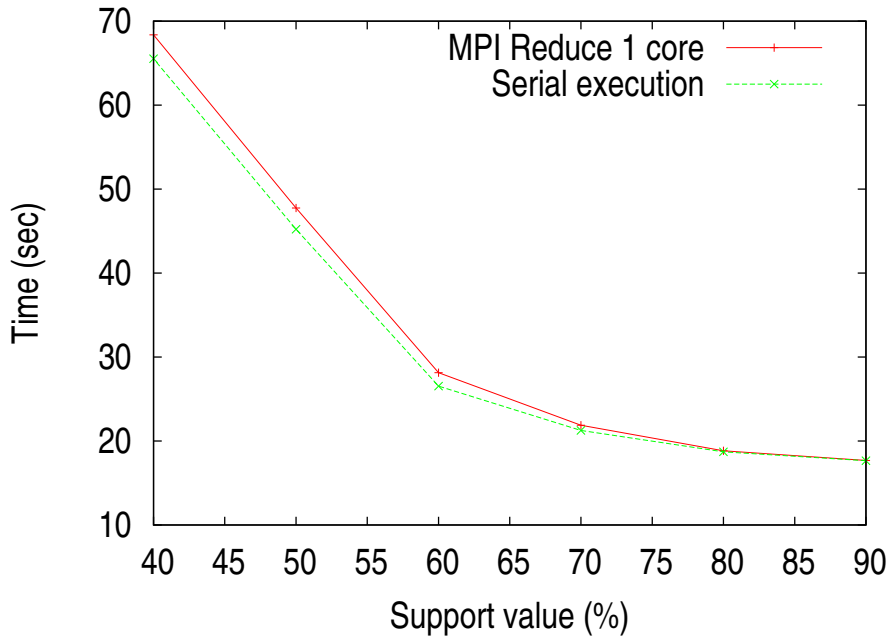


(c) Running on 4, 8, 12 and 16 cores

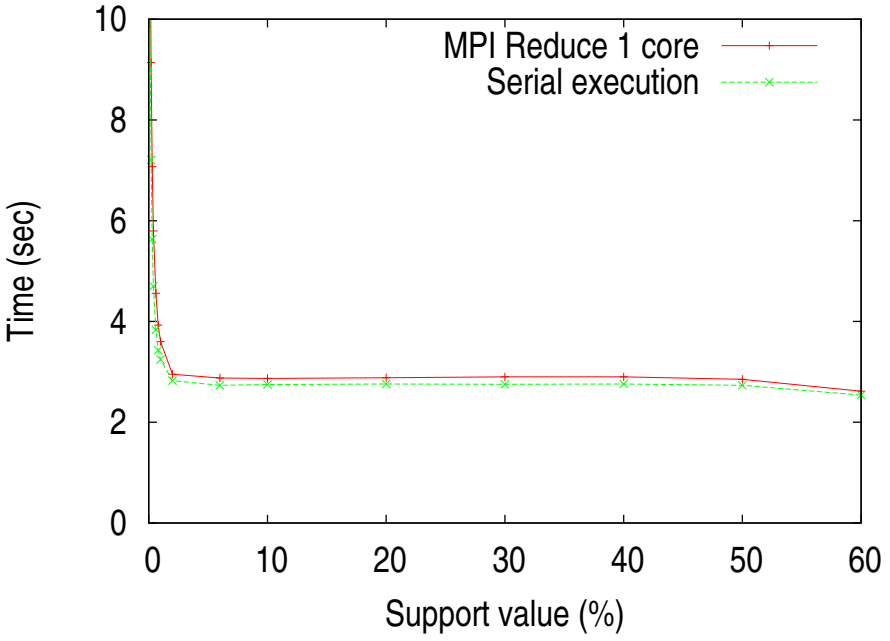


(d) Running on 4, 8, 12 and 16 cores, detailed

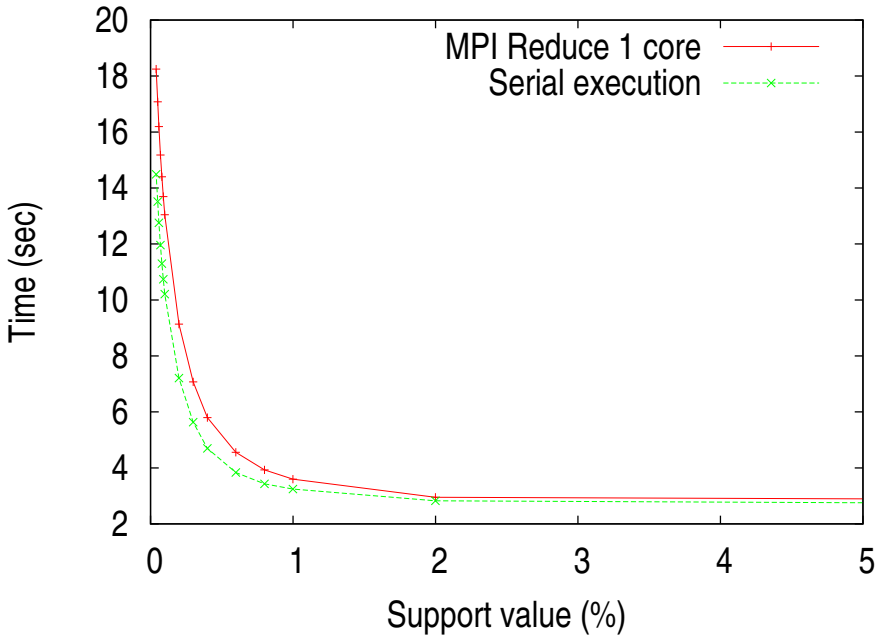
Figure 5.3: Execution times using the Retail dataset



(a) Using the Quest dataset



(b) Using the Retail dataset



(c) Using the Retail dataset, detailed

Figure 5.4: MPI library overhead

Chapter 6

CONCLUSION

In this last chapter of the thesis, an interpretation of the results presented in the previous chapter is given, providing conclusions and discussing possible future work which would be beneficial.

Before we begin with our analysis, it seems important to note a word of caution. Many parameters have an effect on the performance test results shown in the previous chapter, some due to the nature of the data mining field itself and some shared by other programming problems. It is believed that only a small subset of these variables are tested, and a limited number of methods tried in the research outlined in this thesis. Different types of data, different MPI implementations and MPI function combinations, underlying network interconnection infrastructure, data mining application needs and expectations (for example, what support values are more likely to be used?) are just some of the factors to keep in mind.

On the other hand, previous research guides us in what should be expected from a parallel algorithm. Amdahl's Law [21] and Gustafson's Law [42] provide us with a framework to employ. It is known that parts of the work that cannot be parallelized is directly added to the execution time, and so the performance gains are almost never linear. When the processor count is doubled, the time it takes to run do not halve because of the serial parts. Gustafson's Law paints a rosier picture by observing that as problem sizes get larger, parallelism opportunities also increase.

Before the tests had been conducted, there were some expectations and predictions on how the implementation would behave. Particularly, it was expected that the lowering of the support value or adding more processors to the resources would increase efficiency. This idea is based on the common knowledge that parallel programming works best for problems requiring huge processing power. As the support value is lowered, needed processing power

also increases. So, in the usual line of thinking, one could expect that the efficiency of the parallel program would increase. After all, needed computations could be done in a fast parallel way and that would increase the speedup relative to the serial program. Overhead caused by process maintenance and other MPI work would also be compensated easier. Unfortunately, to our surprise, this has not always been the case. The important reason for this is the problem of partial results. As computational requirements increase, so does the partial results that need to be communicated between parallel processes. This higher communication needs put more burden to the processes. In the end, efficiency is hit. As more processes are added or when the support threshold is lowered, speedup increases get smaller, and so efficiency decreases.

This problem of increasing partial results along with the computation required is a general problem faced by other algorithms in the literature. A reason for this is the fact that the best way to increase the computations needed is to increase the data to be processed. But this method mainly results in larger set of results, or as in the parallel programming case, larger communication overhead. With an optimal number of processors, gains in parallel processing would overweight the loss in communication overhead. This optimal value would depend on data and the field of work, the way data is going to be used. Finding this optimal value and deciding on how to use a parallel programming solution may become a challenging, but nonetheless required task on its own.

Our solution presented in this thesis, would be best suited for applications where a fast response is needed for frequent itemsets having medium to high support values. One application area might be the web where fast responses are needed and interesting frequent itemsets having high support values are requested. In this case, more computing power might be added in terms of processors and faster results are obtained. On the other hand, if a very high number of frequent itemsets are sought after for, by using a low support threshold value, then the serial version of the algorithm or other alternatives must be considered.

As described previously, communication costs can become a high burden especially for more sparse datasets where the data to be shared cannot be compressed as in the denser datasets. Different remedies for the communication burden can be developed and tested. Some of the solutions which could be used will now be given as an advise for future work. A straight forward solution is using a more compact data representation in terms of programming language constructs. Data type could be stripped to hold only the barely necessary data and not waste any extra bytes. A natural extension to this could

be modifying the data representation structure for further compression. On the other hand, an existing compression algorithm could also be used independent of the data type or programming language constructs used. These solutions would make the project more complex, but it can prove to be useful for a much wanted performance boost.

Another approach for minimizing communication costs is changing the underlying network interface or using a different MPI implementation. Open MPI is not seen as the fastest MPI implementation, there have been various complaints about its inefficiency especially regarding data communication. It could be interesting to compare how mpich/mvapich and other MPI libraries perform with respect to Open MPI. Another variable which could be tested is the network connectivity methods. In Open MPI, the most advanced connectivity is used as default and this is InfiniBand in our case. Using another connectivity method, such as Gigabit Ethernet could prove to be beneficial. Experimenting with other MPI implementation and/or network interconnectivity could provide an insight for the best environment for the project.

More detailed CPU architecture analysis can also be beneficial for performance improvement. Cache friendliness and newer instruction set extensions could be looked into for possible uses. These methods have been used in previous research and reported to provide high improvements when executed correctly.

A different and more challenging area to be researched is the algorithm. It is believed that the parallel CLOSET+ algorithm discussed in this thesis has potential for development. In particular, performance and efficiency could be increased vastly if duplications of results in different processes could be eliminated. This would require a more fine grained division of the FP-tree representing the database than the item based approach employed in our algorithm. Second part of our algorithm could still be used to merge result trees. There would most certainly be an overhead in generating a more fine grained work division, it is left to the researcher to decide if the speedup gains resulting from a better division would compensate this overhead.

It is envisaged by the author that data mining field is still a young field on its way to higher prominence. This idea is based on the fact that data available is increasing exponentially, with the help of computerization and the Internet. On the other hand, parallel programming is also gaining momentum as clusters and multi-core processors become common. In this thesis a parallel version of a popular frequent itemset mining algorithm CLOSET+ has been presented. This research is hoped to be useful for others studying the exciting fields of data mining and parallel computing.

Bibliography

[1] "3 Billion" post on Flickr blog. http://blog.flickr.net/en/2008/11/03/3-billion/. Retrieved on March 2009.

[2] An STL Error Message Decryptor for C++. http://www.bdsoft.com/tools/stlfilt.html. Retrieved on April 2009.

[3] Boost C++ Libraries. http://www.boost.org/. Retrieved on April 2009.

[4] CERN FAQ - LHC The Guide. http://cdsmedia.cern.ch/img/CERN-Brochure-2008-001-Eng.pdf. Retrieved on March 2009.

[5] Debugging Parallel Programs (in Turkish - Koşut Uygulamalarda Hata Ayıklama) presentation given in the BAŞARIM09 conference. http://www.slideshare.net/tayfun/hata-ayiklama. Retrieved on April 2009.

[6] Explore interesting content around Flickr. http://www.flickr.com/explore/interesting/. Retrieved on March 2009.

[7] Google Flu Trends. http://www.google.org/flutrends/. Retrieved on March 2009.

[8] Homepage of the popular text editor. http://www.vim.org/. Retrieved on April 2009.

[9] IBM Quest Synthetic Data Generator. http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html. Retrieved on March 2009.

[10] Interestingness ranking of media objects. Patent Application 0060242139. http://appft1.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PG01&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.html&r=1&f=G&l=50&s1=%2220060242139%22.PGNR.&OS=DN/20060242139&RS=DN/20060242139. Retrieved on March 2009.

[11] Netcraft March 2009 Web Server Survey. http://news.netcraft.com/archives/2009/03/15/march_2009_web_server_survey.html. Retrieved on March 2009.

[12] Open MPI: A High Performance Message Passing Library. http://www.open-mpi.org/. Retrieved on April 2009.

[13] OpenMP Execution model image taken from Wikipedia, under the terms of GNU Free Documentation Licence. http://en.wikipedia.org/wiki/File:Fork_join.svg. Retrieved on April 2009.

[14] Templatized C++ Command Line Parser Library. http://tclap.sourceforge.net/. Retrieved on March 2009.

[15] Ramesh C. Agarwal, Charu C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *J. Parallel Distrib. Comput.*, 61(3):350–371, 2001.

[16] Dakshi Agrawal and Charu C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 247–255, New York, NY, USA, 2001.

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

[18] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993.

[19] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994.

[20] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. *SIGMOD Rec.*, 29(2):439–450, 2000.

[21] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967.

[22] Amnon Barak and Oren La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4-5):361 – 372, 1998. HPCN '97.

[23] L. A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[24] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming Distributed Memory Sytems Using OpenMP. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, March 2007.

[25] Roberto J. Bayardo, Jr. Efficiently mining long patterns from databases. *SIGMOD Rec.*, 27(2):85–93, 1998.

[26] Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: a case study. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 254–260, New York, NY, USA, 1999.

[27] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26(2):255–264, 1997.

[28] Aaron Ceglar and John F. Roddick. Association mining. *ACM Comput. Surv.*, 38(2):5, 2006.

[29] Dehao Chen, Chunrong Lai, Wei Hu, WenGuang Chen, Yimin Zhang, and Weimin Zheng. Tree partition based parallel frequent pattern mining on shared memory systems. *Parallel and Distributed Processing Symposium, International*, 0:363, 2006.

[30] W. Cheung and O.R. Zaiane. Incremental mining of frequent patterns without candidate generation or support constraint. In *Database Engineering and Applications Symposium, 2003. Proceedings. Seventh International*, pages 111–116, July 2003.

[31] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, Jan-Mar 1998.

[32] Frederica Darema. The spmd model: Past, present and future. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, page 1, London, UK, 2001.

[33] Christie Ezeife and Yue Su. chapter Mining Incremental Association Rules with Generalized FP-Tree, pages 147–160. 2002.

[34] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 17(3), 1996.

[35] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

[36] Liqiang Geng and Howard J. Hamilton. Interestingness measures for data mining: A survey. *ACM Comput. Surv.*, 38(3), 2006.

[37] Jeremy Ginsberg, Matthew H. Mohebbi, Rajan S. Patel, Lynnette Brammer, Mark S. Smolinski, and Larry Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457(7232):1012–1014, Feb 2009.

[38] B. Goethals. Survey on frequent pattern mining, 2003.

[39] Paul Graham. A Plan for Spam. `http://www.paulgraham.com/spam.html`. Retrieved on March 2009.

[40] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the ICDM Workshop on Frequent Itemset Mining*, 2003.

[41] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789 – 828, 1996.

[42] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[43] Eui-Hong Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *Knowledge and Data Engineering, IEEE Transactions on*, 12(3):337–352, May/Jun 2000.

[44] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1):55–86, 2007.

[45] Jiawei Han and Micheline Kamber. *Data Mining, Second Edition: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, January 2006.

[46] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, 2000.

[47] Brian Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[48] Asif Javed and Ashfaq Khokhar. Frequent pattern mining on message passing multiprocessor systems. *Distrib. Parallel Databases*, 16(3):321–334, 2004.

[49] Wenmin Li, Jiawei Han, and Jian Pei. Cmar: Accurate and efficient classification based on multiple class-association rules. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 369–376, Washington, DC, USA, 2001.

[50] Guimei Liu, Hongjun Lu, Jeffrey Xu Yu, Wei Wang, and Xiangye Xiao. Afopt: An efficient implementation of pattern growth approach. In *Proceedings of the ICDM workshop on frequent itemset mining*, 2003.

[51] Li Liu, Eric Li, Yimin Zhang, and Zhizhong Tang. Optimization of frequent itemset mining on multiple-core processor. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1275–1285, 2007.

[52] R. Lottiaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot. OpenMosix, OpenSSI and Kerrighed: a comparative study. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, pages 1016–1023 Vol. 2, May 2005.

[53] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. Parallel mining of frequent closed patterns: Harnessing modern computer architectures. In *ICDM '07: Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, pages 242–251, Washington, DC, USA, 2007.

[54] Gordon Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[55] Rishiyur S. Nikhil and Arvind. *Implicit parallel programming in pH*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[56] Hájek P., Havel I., and Chytil M. The GUHA method of automatic hypotheses determination. *Computing*, (1):293–308, 1966.

[57] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 175–186, New York, NY, USA, 1995.

[58] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. Efficient parallel data mining for association rules. In *CIKM '95: Proceedings of the fourth international conference on Information and knowledge management*, pages 31–36, New York, NY, USA, 1995.

[59] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 398–416, London, UK, 1999.

[60] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: hyper-structure mining of frequent patterns in large databases. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 441–448, 2001.

[61] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *Proc. 2000 ACM-SIGMOD Int. Workshop Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[62] G.F. Pfister. The varieties of single system image. In *Advances in Parallel and Distributed Systems, 1993, Proceedings of the IEEE Workshop on*, pages 59–63, Oct 1993.

[63] Gregory Piatetsky-Shapiro. Knowledge Discovery in Real Databases: A Report on the IJCAI-89 Workshop. *AI Magazine*, 11(5):68 – 70, January 1991.

[64] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. *SIGMOD Rec.*, 27(2):343–354, 1998.

[65] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB '95: Proceedings*

of the 21th International Conference on Very Large Data Bases, pages 432–444, San Francisco, CA, USA, 1995.

[66] J. Ben Schafer, Joseph Konstan, and John Riedi. Recommender systems in e-commerce. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 158–166, New York, NY, USA, 1999.

[67] Toby Segaran. *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. O'Reilly Media, Inc., August 2007.

[68] J. Seward, N. Nethercote, J. Weidendorfer, and the Valgrind Development Team. *Valgrind 3.3 - Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory Ltd., 2008.

[69] Richard Stallman, Roland Pesch, Stan Shebs, and et al. *Debugging with GDB*. Free Software Foundation, 2006.

[70] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[71] Domenico Talia. Parallelism in knowledge discovery techniques. In *PARA '02: Proceedings of the 6th International Conference on Applied Parallel Computing Advanced Scientific Computing*, pages 127–138, London, UK, 2002.

[72] Pang-Ning Tan, Vipin Kumar, and Jaideep Srivastava. Selecting the right interestingness measure for association patterns. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 32–41, New York, NY, USA, 2002.

[73] W. Tansey and E. Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[74] Hannu Toivonen. Sampling large databases for association rules. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 134–145, San Francisco, CA, USA, 1996.

[75] Jaideep Vaidya and Chris Clifton. Privacy preserving association rule mining in vertically partitioned data. In *KDD '02: Proceedings of the eighth ACM SIGKDD*

international conference on Knowledge discovery and data mining, pages 639–644, New York, NY, USA, 2002.

[76] Vassilios S. Verykios, Elisa Bertino, Igor Nai Fovino, Loredana Parasiliti Provenza, Yucel Saygin, and Yannis Theodoridis. State-of-the-art in privacy preserving data mining. *SIGMOD Rec.*, 33(1):50–57, 2004.

[77] Jianyong Wang, Jiawei Han, and Jian Pei. CLOSET+: searching for the best strategies for mining frequent closed itemsets. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 236–245, New York, NY, USA, 2003.

[78] Osmar R. Zaïane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *ICDM '01: Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 665–668, Washington, DC, USA, 2001.

[79] Mohammed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, pages 343–373, December 1997.

[80] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowl. and Data Eng.*, 12(3):372–390, 2000.

[81] Mohammed J. Zaki and Ching jui Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proc. of the 2002 SIAM international conference on data mining (SDM'02)*, pages 457–473, 2002.

[82] Andreas Zeller and Dorothea Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.

[83] Feida Zhu, Xifeng Yan, Jiawei Han, Philip S. Yu, and Hong Cheng. Mining colossal frequent patterns by core pattern fusion. *Data Engineering, International Conference on*, 0:706–715, 2007.