PARALLEL MINING OF MINIMAL SAMPLE UNIQUE ITEMSETS

A dissertation submitted to The University of Manchester for the degree of MSc in Software Engineering in the Faculty of Engineering and Physical Sciences

YEAR OF SUBMISSION
2006-2007

PARASKEVAS YIAPANIS

School of Computer Science
**List of Contents**

**Chapter 1: Introduction**
- 1.1 Statistical Disclosure Control 14
- 1.2 Data Mining 16
  - 1.2.1 Towards Knowledge Discovery 17
- 1.3 Dissertation Objectives 19
- 1.4 Dissertation Overview 20

**Chapter 2: Frequent Itemset Mining**
- 2.1 Introduction 21
- 2.2 The Apriori Algorithm 22
- 2.3 Mining Frequent Itemsets without Candidate Generation 24
- 2.4 Maximal Mining of Frequent Itemsets 30
- 2.5 Parallel Mining of Frequent Itemsets 34
- 2.6 Summary 35

**Chapter 3: Parallel Frequent Itemset Mining**
- 3.1 Introduction 36
- 3.2 The Count Distribution Algorithm 37
- 3.3 The Data Distribution Algorithm 38
- 3.4 The Candidate Distribution Algorithm 39
- 3.5 Parallel Mining of Frequent Itemsets without Candidate Generation 40
- 3.6 Parallel Mining of Maximal Frequent Itemsets 44
- 3.7 Summary 46

**Chapter 4: Related Work**
- 4.1 Introduction 47
- 4.2 $k$-anonymity 48
  - 4.2.1 Notation and Objectives 48
  - 4.2.2 $k$-anonymity Principle 48
  - 4.2.3 Generalization and Suppression 49
  - 4.2.4 Minimal Generalization with Suppression 51
  - 4.2.5 $k$-anonymity Attacks 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.6 k-anonymity Weaknesses</td>
<td>57</td>
</tr>
<tr>
<td>4.3 Mining Minimal Infrequent Itemsets</td>
<td>58</td>
</tr>
<tr>
<td>4.4 Summary</td>
<td>60</td>
</tr>
<tr>
<td><strong>Chapter 5: Sequential Program Design</strong></td>
<td>61</td>
</tr>
<tr>
<td>5.1 Depth-First Search Vs Breadth-First Search</td>
<td>61</td>
</tr>
<tr>
<td>5.2 Mining for Minimal Unique Itemsets in Microdata</td>
<td>62</td>
</tr>
<tr>
<td>5.3 SUDA – Special Unique Detection Algorithm</td>
<td>63</td>
</tr>
<tr>
<td>5.3.1 Example</td>
<td>65</td>
</tr>
<tr>
<td>5.4 A New Recursive Approach – SUDA2</td>
<td>67</td>
</tr>
<tr>
<td>5.4.1 Search Space Representation</td>
<td>67</td>
</tr>
<tr>
<td>5.4.2 Properties of MSUs</td>
<td>69</td>
</tr>
<tr>
<td>5.4.3 Algorithmic Details</td>
<td>69</td>
</tr>
<tr>
<td>5.4.4 Example</td>
<td>71</td>
</tr>
<tr>
<td>5.4.5 Performance Accomplishments</td>
<td>75</td>
</tr>
<tr>
<td>5.5 Summary</td>
<td>75</td>
</tr>
<tr>
<td><strong>Chapter 6: Parallel Program Design</strong></td>
<td>76</td>
</tr>
<tr>
<td>6.1 Parallel Computing and Parallel Programming</td>
<td>76</td>
</tr>
<tr>
<td>6.2 The Message Passing Paradigm</td>
<td>78</td>
</tr>
<tr>
<td>6.3 Message Passing Interface – MPI</td>
<td>79</td>
</tr>
<tr>
<td>6.4 Parallel SUDA2</td>
<td>80</td>
</tr>
<tr>
<td>6.4.1 Parallel Design Issues</td>
<td>80</td>
</tr>
<tr>
<td>6.4.2 SUDA2: A Divide-and-Conquer Algorithm</td>
<td>82</td>
</tr>
<tr>
<td>6.4.3 SUDA2-Specific Parallel Issues</td>
<td>83</td>
</tr>
<tr>
<td>6.4.4 Implementation</td>
<td>84</td>
</tr>
<tr>
<td>6.5 Summary</td>
<td>90</td>
</tr>
<tr>
<td><strong>Chapter 7: Performance Evaluation</strong></td>
<td>91</td>
</tr>
<tr>
<td>7.1 Introduction</td>
<td>91</td>
</tr>
<tr>
<td>7.2 Sources of Overhead in Parallel Programs</td>
<td>91</td>
</tr>
<tr>
<td>7.3 Performance Metrics for Parallel Systems</td>
<td>93</td>
</tr>
<tr>
<td>7.3.1 Execution Time</td>
<td>93</td>
</tr>
<tr>
<td>7.3.2 Speed-up</td>
<td>93</td>
</tr>
</tbody>
</table>
7.3.3 Efficiency 94
7.4 Scalability of Parallel Systems 95
7.5 Datasets Used in Experiments 96
  7.5.1 Connect-4 Dataset 96
  7.5.2 PUMS Dataset 97
  7.5.3 Synthetic Transactional Dataset 97
7.6 Experimental Results 97
7.7 Performance Analysis 98
  7.7.1 Analysis using Connect-4 dataset with $maxk = 5$ 100
  7.7.2 Analysis using Connect-4 dataset with $maxk = 12$ 104
  7.7.3 Analysis using PUMS dataset with $maxk = 4$ 109
  7.7.4 Analysis using Synthetic transactional dataset with $maxk = 5$ 114
  7.7.5 Comparison using Connect-4 dataset with $maxk = 5$ and $maxk = 12$ 118
7.8 Summary 120

Chapter 8: Conclusions 121
8.1 Summary and Findings 121
  8.1.1 Scalability 122
  8.1.2 Efficiency 124
8.2 Future Work 126
  8.2.1 SUDA2-specific load balancing enhancements 126
  8.2.2 Dynamic load balancing schemes 128
  8.2.3 Producing dynamic work generation 129
  8.2.4 Performance analysis 131
  8.2.5 Experiment with message size reduction 133
  8.2.6 Experiment with reduction in database scans 134
  8.2.7 Implementation of SUDA2 using POSIX Threads 136
  8.2.8 Overlapping Computation and Communication 137
  8.2.9 Comparison with other infrastructures 138
8.3 Summary 138
## List of Figures

**Chapter 1: Introduction**  
Figure 1.1: Example of re-identifying individuals by linking to external data 16  
Figure 1.2: An overview of the steps in the KDD process 17  

**Chapter 2: Frequent Itemset Mining**  
Figure 2.1: The Apriori pseudo-code 23  
Figure 2.2: The FP-tree construction method 25  
Figure 2.3: A sample of transactional data 26  
Figure 2.4: The FP-tree constructed from transactional data 27  
Figure 2.5: The algorithm for mining the FP-tree 28  
Figure 2.6: Information obtained from mining the FP-tree 29  
Figure 2.7: The conditional FP-tree associated with node I3 29  
Figure 2.8: A sample of a complete subset lattice for four items 31  
Figure 2.9: The Parent Equivalence Pruning method 32  
Figure 2.10: The Frequent Head Union Tail method 32  
Figure 2.11: The Head Union Tail MFI method 33  
Figure 2.12: The complete MAFIA pseudo code 34  

**Chapter 3: Parallel Frequent Itemset Mining**  
Figure: 3.1: The Count Distribution Algorithm 38  
Figure: 3.2: The Data Distribution Algorithm 39  
Figure: 3.3: Construction of local FP-trees from local database partitions over 2 processors \((P_1, P_2)\) 42  
Figure: 3.4: Mining frequent itemsets in parallel \((P_0)\) 43  
Figure: 3.5: Mining frequent itemsets in parallel \((P_1)\) 43  

**Chapter 4: Related Work**  
Figure: 4.1: Example of \(k\)-anonymity, where \(k = 2\) and \(Q_T = \{\text{Race, Birth, Gender, ZIP}\}\) 50  
Figure: 4.2: Example of a dataset before suppression is applied 50
Figure: 4.3: Example of a dataset after suppression is applied 51
Figure: 4.4: Example of a Private Table $PT$ and its generalized tables 52
Figure: 4.5: Example of generalized tables, with suppression for the Private Table $PT$ of Figure 4.4 53
Figure: 4.6: Example unsorted matching attack 54
Figure: 4.7: A private table $PT$ before release 55
Figure: 4.8: $GT_{[1]}$ A generalized table for the private table in Figure 4.7 55
Figure: 4.9: $GT_{[2]}$ A generalized table for the private table in Figure 4.7 56
Figure: 4.10: A linking table $LT$ generated from $GT_{[1]}$ and $GT_{[2]}$ 56
Figure: 4.11: Pseudo code for MINIT($D,[1:L], maxk$) 59

Chapter 5: Sequential Program Design 61
Figure: 5.1: A tree structure consisting of nine nodes 61
Figure: 5.2: Example of a dataset 65
Figure: 5.3: The sorting process that SUDA accommodates 66
Figure: 5.4: The process of searching for MSUs 68
Figure: 5.5: Pseudo code for SUDA2 71
Figure: 5.6: Table $T$: 6 rows, 5 columns 72
Figure: 5.7: Sub-table $TA_2$ 72
Figure: 5.8: Sub-table $TE_{2C1}$ is extracted from $TE_2$ 73
Figure: 5.9: Sub-table $TE_{3C1}$ is extracted from $TE_3$ 74
Figure: 5.10: Analyzing and breakdown process for sub-table $TA_1$ 74

Chapter 6: Parallel Program Design 76
Figure: 6.1: A summary of the minimal-set of MPI functions 79
Figure: 6.2: An example of level 1 work generation 85
Figure: 6.3: Distribution of jobs for level 1 work generation in parallel SUDA2 86
Figure: 6.4: Sub-table $TE_{2C1}$ is extracted from $TE_2$ 87
Figure: 6.5: An example of a work generation based on workload 89
Chapter 7: Performance Evaluation

Figure: 7.1: Execution time for Connect-4 dataset with $maxk = 5$ 99
Figure: 7.2: Speedup for Connect-4 dataset with $maxk = 5$ 99
Figure: 7.3: Efficiency for Connect-4 dataset with $maxk = 5$ 99
Figure: 7.4: Speedup results for Connect-4 with $maxk = 5$ 100
Figure: 7.5: Efficiency results for Connect-4 with $maxk = 5$ 101
Figure: 7.6: Execution time for Connect-4 dataset with $maxk = 12$ 103
Figure: 7.7: Speedup for Connect-4 dataset with $maxk = 12$ 103
Figure: 7.8: Efficiency for Connect-4 dataset with $maxk = 12$ 103
Figure: 7.9: Speedup results for Connect-4 with $maxk = 12$ 105
Figure: 7.10: Efficiency results for Connect-4 with $maxk = 12$ 106
Figure: 7.11: Execution time for PUMS dataset with $maxk = 4$ 108
Figure: 7.12: Speedup for PUMS dataset with $maxk = 4$ 108
Figure: 7.13: Efficiency for PUMS dataset with $maxk = 4$ 108
Figure: 7.14: Speedup results for PUMS with $maxk = 4$ 109
Figure: 7.15: Efficiency results for PUMS with $maxk = 4$ 110
Figure: 7.16: Execution time for Synthetic transactional dataset with $maxk = 5$ 113
Figure: 7.17: Speedup for Synthetic transactional dataset with $maxk = 5$ 113
Figure: 7.18: Efficiency for Synthetic transactional dataset with $maxk = 5$ 113
Figure: 7.19: Speedup results for Synthetic transactional with $maxk = 5$ 114
Figure: 7.20: Efficiency results for Synthetic transactional with $maxk = 5$ 116
Figure: 7.21: Speedup comparison for Connect-4 dataset 118
Figure: 7.22: Efficiency comparison for Connect-4 dataset 118

Chapter 8: Conclusions

Figure: 8.1: Speedup comparison for Connect-4 dataset with $maxk = 5$ 123
Figure: 8.2: Speedup comparison for Synthetic transactional dataset with $maxk = 5$ 123
Figure: 8.3: Efficiency comparison for Connect-4 dataset with $maxk = 12$ 125
Figure: 8.4: Efficiency comparison for Synthetic transactional dataset with \( maxk = 5 \)

Figure: 8.5: The separator producing work generation as a step function

Figure: 8.6: Efficiency comparison for Synthetic transactional dataset with \( maxk = 5 \)

Figure: 8.7: The vertical data representation version of the dataset in Figure 2.3
Abstract

Information release for any purpose requires that all records are provided under guarantees of confidentiality. However removal of sensitive attributes such as name and social security number, are not sufficient to ensure this. This problem is known as Statistical Disclosure. A unique combination of attributes can be used to re-identify individuals’ information which is supposed to remain confidential. We refer to such attributes as Minimal Unique Itemsets. The practice of identifying such itemsets is important in providing Statistical Disclosure Control.

This dissertation is concerned with the investigation of parallel solutions for the recursive search algorithm for minimal unique itemsets, SUDA2. This algorithm is a powerful tool for providing statistical disclosure control. We show how data mining techniques inspired ideas for the creation of the algorithm and what reasons gave motivation for the design of SUDA2. The parallel implementation, as well as the sequential version of the algorithm, is discussed extensively. The parallel implementation adapts the algorithm to a cluster of computers allowing work to be shared among multiple processors using the message-passing paradigm. The nature of SUDA2 allows work to be divided into non-overlapping tasks enabling high performance achievements.

Although parallel SUDA2 is several orders of magnitude faster than the sequential SUDA2 there remains room for improvement as far as load balancing is concerned. For many cases the tasks generated are very fine grained (implying high communication overheads) causing larger tasks to wait last for processing. In such cases the remaining large tasks may not be able to fully utilize the system and thus result in load imbalance.

An optimization method for facilitating load balancing is described and tested with parallel SUDA2. This method imposes limits on the number of sub-tasks to be generated based on their size. The further we descent in the recursion tree, the smaller the tasks become. Thus we only need to select the largest tasks at each level of recursion.
With this new load balancing approach incorporated, the revised algorithm appears to be running twice as fast as before for finer levels of granularity. The Message Passing Interface (MPI) is used in achieving our objectives.

**Key words:** data mining, infrequent patterns, unique itemset, Minimal Sample Uniques, Minimal Infrequent Itemsets, parallel algorithms, statistical disclosure, recursive algorithm, load balancing
Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
COPYRIGHT STATEMENT

Copyright in text of this dissertation rests with the author. Copies (by any process) either full, or of extracts, may be made only in accordance with instructions given by the author. Details may be obtained from the appropriate Graduate Office. This page must be part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.

The ownership of any intellectual property rights which may be described in this dissertation is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available for the Head of the School of Computer Science (or the Vice-President and Dean of the Faculty of Engineering and Physical Sciences).
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my parents for their moral and financial support throughout the years of my studies.

I wish to express my sincere appreciation to my supervisor, Professor John A. Keane, for his intelligent and untiring supervision. I warmly thank him for his guidance.

Furthermore, I would like to thank Dr Ken Mayes and Dr Anna Manning for all their help and cooperation for this dissertation as well as Professor David Haglin for providing access to his High-performance Computer Cluster at MSU (Minnesota State University).

Finally, I would like to thank my friend Agni for proofreading this dissertation.
Chapter 1

Introduction

1.1 Statistical Disclosure Control

During the last years, it has been observed that there is an increasing demand on information release for researchers, local government, health authorities and many others [3]. Unfortunately data sharing can sometimes lead to serious problems associated with disclosure of personal information which should not have been disclosed. Prior to sharing data, the data holders must eliminate all fields that may possibly enable identification of an individual. Such fields are referred to as sensitive data and include data, which can be used directly to identify individuals, such as Social Security Number (SSN), name, etc. On the other hand, data that cannot directly reveal the identity of an individual (i.e. birth date, gender, ZIP code, etc) is referred to as non-sensitive data [14].

However, the removing of the sensitive data is inadequate to address the problem. A combination of non-sensitive data can be linked with external data and possibly re-identify individuals which are supposed to be kept anonymous. Such combinations of non-sensitive attributes are referred to as quasi-identifiers. As data is released for several purposes, the importance of each data varies depending on the case. If
somebody can have access to more than one (related) file, he/she may possibly use the quasi-identifiers (regarded as of no importance) to uncover identities. This problem is known as *Statistical Disclosure* [18]. This highlights the need to develop new effective ways to protect information from being disclosed.

A set of attributes either being quasi-identifiers or not, is referred to as an *itemset*. An itemset that appears in exactly one row of the dataset is known as *unique* itemset and is usually formed by a combination of unusual attributes (e.g. a 16-year-old widow).

A unique itemset with no unique proper subsets is known as *minimal unique itemset (MUI)* or *minimal sample unique (MSU)* [3, 4, 12]. Identifying MSUs is a critical task for Statistical Disclosure Control as it is possible to match directly an MSU with a record in another dataset and reveal information which is supposed to be confidential. The smaller (in attribute size) the MSU and the greater the number of MSUs in a given record, the higher the disclosure risk that record has [3]. To illustrate the problem Figure 1.1 exemplifies a table of medical data to be released.

This table is released in a form that it is supposed to be confidential. However, some of the attributes contained within the medical table may also appear in another external table (e.g. one containing voters list).

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Race</th>
<th>Birth</th>
<th>Gender</th>
<th>ZIP</th>
<th>Marital Status</th>
<th>Medical Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asian 09/27/64</td>
<td>Female 94139</td>
<td>Divorced</td>
<td>Hypertension</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian 09/30/64</td>
<td>Female 94139</td>
<td>Divorced</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian 04/18/64</td>
<td>Male 94139</td>
<td>Married</td>
<td>Chest pain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asian 04/15/64</td>
<td>Male 94139</td>
<td>Married</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black 03/13/63</td>
<td>Male 94138</td>
<td>Married</td>
<td>Hypertension</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black 03/18/63</td>
<td>Male 94138</td>
<td>Married</td>
<td>Short breath</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black 09/13/64</td>
<td>Female 94141</td>
<td>Married</td>
<td>Short breath</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Black 09/07/64</td>
<td>Female 94141</td>
<td>Married</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White 05/14/61</td>
<td>Male 94138</td>
<td>Single</td>
<td>Chest pain</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White 05/08/61</td>
<td>Male 94138</td>
<td>Single</td>
<td>Obesity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>White 09/15/61</td>
<td>Female 94142</td>
<td>Widow</td>
<td>Short breath</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Voter List

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>City</th>
<th>ZIP</th>
<th>Birth</th>
<th>Gender</th>
<th>Party</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haris Georgiou</td>
<td>12 Iakovou St</td>
<td>San</td>
<td>Francisco</td>
<td>94142</td>
<td>9/15/61</td>
<td>Female</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: Example of re-identifying individuals by linking to external data

It is possible for someone to link data from both tables and uncover sensitive information about individuals [14]. For instance, in the medical table there is only one female that was born in 09/15/61 and lives in the 94142 area. In this case, this combination is unique in some external data forming a voter list and by linking both tables we identify that female as being “Haris Georgiou, 12 Iakovou St, San Francisco”. Consequently, this reveals that Haris Georgiou is suffering from shortness of breath.

1.2 Data Mining

Organizations rely strongly on Database Systems (DBs) in order to store, retrieve, and manipulate data related to their customers. This creates an ever increasing demand for storage and speed.

Data Mining is “the nontrivial extraction of implicit, previously unknown, and potentially useful information from data” [22]. Data Mining goes even further than merely storing data in a meaningful way and retrieving it. It is sometimes referred to as “KDD” or “Knowledge Discovery from Data”, as it also takes advantage of that data in order to discover some previously unknown “knowledge” from that data. This can be ultimately used to increase customer relationship management, increase productivity or sales, or even facilitate law enforcement [43, 44].
1.2.1 Towards knowledge discovery

In order to discover knowledge from data there is a typical process consisting of a series of steps that needs to be followed [7]:

1. Data Cleaning
2. Data integration
3. Data transformation
4. Data Reduction
5. Data Mining
6. Pattern evaluation
7. Knowledge presentation

The above steps are also illustrated in Figure 1.2. It is also important to note that the KDD process is an iterative process allowing for dynamic changes while shifting from one step to another.

Figure 1.2: An overview of the steps in the KDD process
**Data Preprocessing:** The first four steps basically form the data preprocessing part of the process. Data preprocessing is essentially processing your data in order to improve its quality and ultimately increase the efficiency and ease of the data mining process. Data Cleaning is the process of handling missing values, outliers, noisy data, and inconsistencies. Data Integration deals with linking data coming from multiple (probably inconsistent) sources. Data transformation basically performs modifications in data in order to bring it in forms appropriate for Data Mining. Finally, Data Reduction attempts to accomplish a smaller version of the data, yet allowing the results to be maintained in the same levels of accuracy. All these preprocessing are essential as they assist the Data Mining methods in producing the finest results possible [7, 24].

**Data Mining:** When people refer to Data Mining they mean usually the entire process. However, it is just the most significant step of the entire process. In the Data Mining step, intelligent methods are used in order to extract hidden patterns. In order to perform Data Mining on a given data, a number of methods exist. The most important include: *Association Rules, Classification and Prediction, and Cluster Analysis* [7, 24].

*Association rules* are implication rules used mainly for market basket analysis when we are searching for items that were bought together [5, 8]. In this case why are interested in the strengths of the discovered association rules.

The next chapter focuses on Association Rules, especially with the more common practice of frequent pattern mining. The ideas behind frequent patterns mining will facilitate to the design of algorithms for mining infrequent patterns.

*Classification and Prediction* are two types of data analysis that can be applied to extract models describing important classes or to predict future trends. The main difference between the two is that classification predicts categorical (discrete, unordered) labels whereas prediction models continuous-valued functions.

*Cluster Analysis* organizes the data in a way that enables the gathering of similar objects together in clusters (as implied by its name). Whereas objects of the same cluster are similar to one another, objects from different clusters are dissimilar between them. The main difference between Cluster Analysis and Classification and
Prediction is that the former operates on data objects without consulting a known class label.

The Data Mining process can extract a large number of patterns. It does not necessarily mean that all those patterns are actually useful. Some of them may only present common knowledge or even lack of novelty. *Pattern evaluation* is just deciding which of those patterns are interesting for our purposes [7].

The last step of the process consists of techniques used in order to present the results to the user. These may include representation techniques such as trees, tables, graphs, matrices, rules, etc.

### 1.3 Dissertation Objectives

The process of identifying MSUs is a highly computationally intensive problem as it is often an iterative process testing all the records of the dataset against massive numbers of combinations. In [12] it has been proven that a comprehensive search for MSUs is an NP-hard problem.

In this paper we study an algorithm that uses a recursive approach to discover and grade all MSUs, in a given dataset, eliminating the possibility of an individual to be re-identified.

As mentioned earlier the nature of this type of problems usually demands great computational power. The goal is to take this algorithm a step further and utilize parallel architectures to speed-up its execution. This project combines ideas behind data mining, especially related with Association Rule Mining, and parallel systems in order to facilitate discovery of attributes that can disclose information, while exploiting as much parallelism as possible for efficiency improvements.

The critical issues that affect efficiency of parallel algorithms are:

- Computation that is performed in parallel
- Communication between processors

Typically, when designing parallel algorithms people focus on reducing the communication between processors and increase the computation amounts.
The aim of this work is to investigate parallel solutions for Mining Minimal Sample Uniques, and try to exploit as much parallelism as possible. Therefore, a parallel program for MSUs discovery is described and analyzed extensively. The program was designed in C++ and implemented on a low-cost/high performance computer cluster. The MPI library was chosen to facilitate implementation as it is expected to run faster within a large multi-processor environment instead of other systems. A new optimization method for assisting efficient load balancing was developed. Several experiments were performed on different datasets in order to evaluate the performance of the parallel system. The parallel system is several orders of magnitude faster than the sequential version of it. However the load balancing appears to be a challenging issue allowing room for improvement. The new optimization technique appears to facilitate load balancing, and in most cases runs faster for finer levels of granularity.

1.4 Dissertation Overview

The rest of the paper is organized as follows: Chapter 2 provides some background information on Frequent Pattern Mining explaining algorithmic properties that are also related with our work. Chapter 3 focuses on parallel version for the sequential algorithms presented in Chapter 2. In Chapter 4 different areas related with the problem along are discussed, such as other ideas for Statistical disclosure Control as well as an algorithm which shares numerous similarities with our algorithm. Chapter 5 is concerned with the presentation of the sequential algorithm we are studying. Chapter 6 presents a parallel solution for the algorithm showed in the previous chapter as well as parallel issues that can affect its performance. Furthermore an optimization method is presented to facilitate load balancing. The results of the experiments are given in Chapter 7. Finally Chapter 8 presents conclusions and future enhancements.
Chapter 2

Frequent Itemset Mining

2.1 Introduction

Mining association rules from sets of items in a dataset was first introduced by [7]. Association rules are the byproduct of the frequent pattern mining which will be discussed later on in this paper. Frequent patterns are patterns that appear frequently in a data set. A dataset is a collection of data usually presented in a tabular form. One of the first examples of this type is known as “Market Basket Analysis” [45]. Market Basket Analysis deals with processing customer transactions (i.e. what items a customer bought together) in order to identify the associations between those items and uncover buying habits that those customers may enjoy.

Association Rules are implication rules of the form \( A \Rightarrow B \), meaning that the presence of an item \( A \) in a transaction (purchase) implies the presence of another item \( B \) [5]. Along with the rule there are also two related numbers: support, and confidence. Support is simply the percentage of transactions (out of the whole data set) that include both items \( A \) and \( B \). Confidence is the percentage that shows how many of those transactions that include \( A \), also include \( B \). For instance, the information that customers who buy a computer also tend to buy a printer at the same time is represented by the following Association Rule:
computer ⇒ printer [support = 2%, confidence = 65%]

Therefore a support of 2% means that 2% of all the transactions show that a computer and a printer are bought together. A confidence of 65% means 65% of the total customers who purchased a computer, have also purchased a printer.

Considering a pattern as frequent requires the satisfaction of both a minimum support threshold and a minimum confidence threshold, usually set by the user [8].

To state the problem area more formally let us assume a set of items \( I = \{I_1, I_2, \ldots, I_m\} \). From now on we will refer to a set of items as an itemset. Also an itemset that contains \( k \) items is called a \( k \)-itemset. Let \( D \), be a database containing transactions where each transaction \( T \) is an itemset such as \( T \subseteq I \). Let \( A \) be an itemset. A transaction \( T \) contains \( A \) if \( A \subseteq T \). An association rule, as stated earlier, is an implication rule of the form \( A \Rightarrow B \), where \( A \subset I \), \( B \subset I \), and \( A \cap B = \emptyset \). The support and confidence can be calculated using the following formulae:

\[
\text{support}(A \Rightarrow B) = \frac{\text{support}(A \cup B)}{\text{support}(A)}
\]

\[
\text{confidence}(A \Rightarrow B) = \frac{\text{support}(B|A)}{\text{support}(A)} = \frac{\text{support}(A \cup B)}{\text{support}(A)}
\]

The process for Association Rule mining has two main steps [7, 8]:

1. Discover all frequent itemsets (i.e. those that appear at least as frequent as the minimum threshold determines)
2. Generate strong association rules from the frequent itemsets (i.e. rules that satisfy minimum support and minimum confidence).

### 2.2 The Apriori Algorithm

One of the first algorithms for mining for frequent patterns was Apriori [5, 8]. The name arises from the fact that the algorithm uses prior knowledge as a pruning
technique in order to increase efficiency and reduce the search space. Using a breadth-first search, Apriori, scans the database in a bottom-up level-wise fashion, generating at each level, sets that are candidates to be frequent itemsets.

An important technique is used to reduce the search space at each level. We refer to this technique as the *Apriori Property* and it states that “All nonempty subsets of a frequent itemset must also be frequent” [7].

That is, in each level all the candidates are checked, and in case their subsets are not all frequent; the candidates are then discarded from the process.

Imagine the itemset \{beer\} that is found to be non frequent. In the next level, any superset generated from that itemset, such as \{beer, bread\}, it is impossible to be frequent as there is at least one item from that itemset that was already discovered to be non frequent. Therefore if an itemset fails to “pass the test”, then all of its supersets will consequently fail to “pass it” as well.

```
Input:
D, a database of transactions
min_sup, the minimum support threshold

Algorithm:
L_1 = find_frequent_1-itemsets(D);
for k = 2; L_{k-1} ≠ ∅; k++ ) {
    C_k = apriori_gen(L_{k-1});
    for each transaction t ∈ D { //scan D for counts
        C_t = subset(C_k, t); //get the subsets of t that are candidates
        for each candidate c ∈ C_t
            c.count++;
    }
    L_k = { c ∈ C_k | c.count ≥ min_sup }
} return L = ∪_k L_k;
```

**Figure 2.1:** The Apriori pseudo-code [7].

Figure 2.1 depicts the Apriori algorithm. As we can see the input consists of a Database D containing the transactions taking part in the mining process. In addition the minimum threshold is provided by the user.
The algorithm proceeds as follows:
First, a full scan of the database identifies all frequent 1-itemsets (i.e. itemsets consisting of only one item and satisfy the minimum threshold). The resulted generated set is indicated by $L_1$.

Then the algorithm following a level-wise search iterates through the database to find $L_2$ (the set of 2-itemsets), $L_3$, and so on, until no more frequent $k$-itemsets can be discovered. In order to find $L_k$, a set of candidate $k$-itemsets, denoted as $C_k$, is generated by taking the Cartesian product of $L_{k-1}$ with itself. The apriori_gen function basically generates $C_k$ making also use of the Apriori Property and removing any unfruitful candidates. Once, all candidates are produced, the database is scanned and for each transaction, a function called subset is used to find all subsets of the transaction that are candidates, accumulating their counts as it iterates. In the final stage all candidates that satisfy the minimum threshold will form $L$, the set of frequent itemsets.

Using this method of testing and generating candidates, the algorithm has significant performance gains in terms of reducing the number of candidates that require checking in order to establish their frequency [5, 7, 8]. However there are still some inefficient points that need to be further considered and analyzed while seeking for solutions.

### 2.3 Mining Frequent Itemsets without Candidate Generation

The major cost of Apriori and Apriori-based algorithms is that they may generate a large number of candidates. Furthermore repeated scanning of the dataset for candidate generation is very expensive [7, 9]. For these reasons, other ideas have been proposed for generating frequent patterns without candidate generation. An interesting algorithm following this approach is called FP-growth [9, 10].

FP-growth compresses the dataset into a tree structure, resulting in reduced scans, and then uses a divide-and-conquer approach to partition the mining process into
smaller parts, which ultimately reduces the search space and avoids the costly candidate generation.

The FP (Frequent Pattern) – growth algorithm can be separated into two phases: The \textit{FP-tree construction} phase and the \textit{FP-growth mining process} phase. The first phase is shown in Figure 2.2.

\begin{boxedverbatim}
\textbf{Input:} \\
\hspace{1em} \textit{D}, a database of transactions \\
\hspace{1em} \textit{min\_sup}, the minimum support threshold \\
\textbf{Method:} \\
\hspace{1em} 1. Scan the \textit{D} once and find all frequent 1-itemsets. Insert those itemsets in a list, \textit{L}, sorted in descending order of support count. \\
\hspace{1em} 2. Create the root of the FP-tree and set it to \textit{null}. \\
\hspace{1em} \textbf{for each} \hspace{0.5em} \textit{transaction} \hspace{0.5em} \textit{t} \in \textit{D} \{ \\
\hspace{1.5em} Select and sort the frequent items in \textit{t} based on the order \textit{L} has. \\
\hspace{1.5em} Let the sorted frequent item list in \textit{t} be \textit{[h|T]}, where \textit{h} is the first element of \textit{the list (the head) and \textit{T} is the remaining of the list (the tail).} \\
\hspace{1.5em} \textbf{insertTree}([\textit{h}|\textit{T}], \textit{t}); \\
\hspace{1.5em} \} \\
\hspace{1em} \textbf{insertTree}([\textit{h}|\textit{T}], \textit{t}) \{ \\
\hspace{1.5em} Let \textit{N} be a child of \textit{t} \\
\hspace{1.5em} Let \textit{item-name} be a field of an item in a transaction \textit{t}, indicating the name of the item \\
\hspace{1.5em} \textbf{if} (\textit{N.item-name} == \textit{h.item-name}) \{ \\
\hspace{2.0em} \textit{N.count}++; \\
\hspace{1.5em} \} \\
\hspace{1.5em} \textbf{else} \{ \\
\hspace{2.0em} create new node \textit{N}; \\
\hspace{2.0em} \textit{N.count} = 1; \\
\hspace{2.0em} \textit{N}’s parent link is linked to \textit{t}; \\
\hspace{2.0em} \textit{N}’s link is linked to nodes with the same \textit{item-name}; \\
\hspace{1.5em} \} \\
\hspace{1.5em} \textbf{if} (\textit{T} is nonempty) \{ \\
\hspace{2.0em} \textbf{insertTree}([\textit{T}], \textit{N}); \\
\hspace{1.5em} \} \\
\}
\end{boxedverbatim}

\textbf{Figure 2.2:} The FP-tree construction method [10].
Illustrating it with the use of an example may enable us to gain a better understanding of the FP growth algorithm.

Each transaction is associated with an identifier referred to as $TID$. Consider applying the FP-growth algorithm on the transactional data shown in Figure 2.3.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>T100</td>
<td>I1, I2, I5</td>
</tr>
<tr>
<td>T200</td>
<td>I2, I4</td>
</tr>
<tr>
<td>T300</td>
<td>I2, I3</td>
</tr>
<tr>
<td>T400</td>
<td>I1, I2, I4</td>
</tr>
<tr>
<td>T500</td>
<td>I1, I3</td>
</tr>
<tr>
<td>T600</td>
<td>I2, I3</td>
</tr>
<tr>
<td>T700</td>
<td>I1, I3</td>
</tr>
<tr>
<td>T800</td>
<td>I1, I2, I3, I5</td>
</tr>
<tr>
<td>T900</td>
<td>I1, I2, I3</td>
</tr>
</tbody>
</table>

**Figure 2.3:** A sample of transactional data

So far we have seen the algorithm that constructs the FP-tree. Using the above list of transactions we will describe how this is performed in practice. The first step is to scan the database and create a list of frequent 1-itemsets, along with their support count, sorted in descending order based on their occurrence frequency. For this example we set the minimum support threshold to be 2. Thus the resulting list is $L = \{\{I2:7\}, \{I1:6\}, \{I3:6\}, \{I4:2\}, \{I5:2\}\}$.

The construction of the FP-tree starts by creating the root of the tree and labeling it with $null$. We then analyze the items in each transaction, preserving the order that $L$ utilizes, creating a branch for each transaction. Thus in this case, the first transaction will make constitute the first branch of the tree having the nodes $<I2:1>$, $<I1:1>$, and $<I5:1>$. I2 is linked to the root, I1 is a child of I2, and I3 is a child of I1. The next transaction leads to the creation of the second branch with two nodes: $<I2>$ and $<I4>$. Since I2 already exists in the tree (i.e. both branches share an identical prefix), duplication of this node is unnecessary.

Instead, we increase the cardinality of the existing node by one and create a node $<I4:1>$ to be linked as a child of the existing node I2 (which now is $<I2:2>$).
Therefore, the idea here is the following: whenever a new branch is to be added to the tree, we must take into consideration nodes that already exist and take the appropriate actions (i.e. increment their count by 1). In addition to the tree structure, a table is also built, to keep track of where each item occurs in the tree. Similarly, each item is linked with each of its occurrences in the tree, (using links called node-links) alleviating this way, tree traversal [10].

What we have managed so far is to completely discard the database from the mining process as any further actions will only involve the FP-tree created. No further scans of the database are performed since the FP-tree structure now contains all the information needed for the mining process. The resulting FP-tree after a full transaction scan of the database is shown in Figure 2.4.

![FP-tree](image_url)

**Figure 2.4:** The FP-tree constructed from transactional data.

At this stage the first part of the process has finished. The second phase, as mentioned earlier, consists of the FP-growth mining part. The algorithm that gives life to this idea is shown in Figure 2.5.
**Input:**
FP-tree constructed before in the first step

**Output:**
The set of frequent patterns

**Method:**
Call function `fpGrowth(FP-tree, null)`.

```java
fpGrowth(Tree, α) {
    if Tree contains a single path T
        for each combination (denoted as β) of the nodes in path T
            generate pattern β ∪ α with support_count = minimum support count
            of nodes in β;
    else for each αᵢ in the header of Tree {
        generate pattern β = αᵢ ∪ α with support_count = αᵢ.support_count;
        construct β’s conditional pattern base and then β’s conditional FP_tree Tree_β;
        if Tree_β ≠ ∅ then
            call fpGrowth(Tree_β, β);
    }
}
```

**Figure 2.5:** The algorithm for mining the FP-tree [7].

The algorithm starts operating on frequent 1-itemset patterns (as an initial *suffix pattern*) and for each one constructs its *conditional pattern base*. A conditional pattern base is a set consisting of nodes that form *prefix* paths that lead to the corresponding suffix pattern [10]. Finally a conditional FP-tree is constructed for that pattern and is mined recursively. The pattern growth is attained by the combination of the suffix pattern with frequent patterns generated from such a conditional FP-tree.

First, we start from the last frequent 1-itemset pattern of the list which in our case is I₅. As illustrated in Figure 2.4, the suffix I₅ can be found in two branches of the tree. The two paths that lead to I₅ are: `<I₂, I₁, I₅:1>` and `<I₂, I₁, I₃, I₅:1>`. Thus the two prefix paths `<I₂, I₁:1>` and `<I₂, I₁, I₃:1>` will compose its conditional pattern base set.

In this case there is only one path that constitutes the conditional FP-tree: `<I₂:2, I₁:2>`. I₃ is not participating as its support count is 1 (i.e. less than the minimum...
threshold). Mining the conditional FP-tree generates the following frequent patterns: \{I2, I5:2\}, \{I1, I5:2\}, \{I2, I1, I5:2\}.

In a similar way I4 produces only one frequent pattern: \{I2, I4:2\}. It is important to notice here that even though I5 comes after I4 in the branch there is no need of considering it in the pattern generation as it was already examined in the previous check. For this reason when we have started operating on frequent 1-itemsets, we have chosen to select them from the least frequent to the most frequent.

A summary of the mining process information is shown in Figure 2.6.

<table>
<thead>
<tr>
<th>Item</th>
<th>Conditional Pattern Base</th>
<th>Conditional FP-tree</th>
<th>Frequent Patterns Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>I5</td>
<td>{{I2, I1:1}, {I2, I1, I3:1}}</td>
<td>\langle I2:2, I1:2\rangle</td>
<td>{I2, I5:2}, {I1, I5:2}, {I2, I1, I5:2}</td>
</tr>
<tr>
<td>I4</td>
<td>{{I2, I1:1}, {I2:1}}</td>
<td>\langle I2:2\rangle</td>
<td>{I2, I4:2}</td>
</tr>
<tr>
<td>I3</td>
<td>{{I2, I1:2}, {I2:2}, {I1:2}}</td>
<td>\langle I2:4, I1:2\rangle, \langle I1:2\rangle</td>
<td>{I2, I3:4}, {I1, I3:4}, {I2, I1, I3:2}</td>
</tr>
<tr>
<td>I1</td>
<td>{{I2:4}}</td>
<td>\langle I2:4\rangle</td>
<td>{I2, I1:4}</td>
</tr>
</tbody>
</table>

Figure 2.6 Information obtained from mining the FP-tree.

Mining I3, produces the conditional base consisting of: \{\{I2, I1:2\}, \{I2:2\}, \{I1:2\}\} and a conditional tree with two branches: \langle I2:4, I1:2\rangle, \langle I1:2\rangle. The tree obtained is shown in Figure 2.7.

Figure 2.7 The conditional FP-tree associated with node I3.
The frequent patterns generated from this tree are: \{I2, I3:4\}, \{I1, I3:4\}, \{I2, I1, I3:2\}.

The same idea is applied for node I1 which generates only one frequent pattern: \{I2, I1:4\}.

Mining frequent patterns without candidate generation reduces substantially the searching cost. A study by [9] shows that this method is about an order of magnitude faster than Apriori algorithm.

### 2.4 Maximal Mining of Frequent Itemsets

While mining without candidate generation can be more efficient than Apriori-based algorithms, it may still be costly in the sense that the process may produce a very large number of frequent itemsets, especially when the minimum support threshold is set very low or when long patterns appear within the dataset [7, 9].

A very useful observation shows that since frequent itemsets are upward closed, hence it is sufficient to discover only the Maximal Frequent Itemsets (MFIs) from a data set [6, 11] An itemset \(X\) is an MFI if \(X\) is frequent and there exists no superset of \(X\) that is frequent. Using this significantly decreases the number of patterns generated while preserving full information concerning the set of frequent itemsets.

An effective algorithm that mines for MFIs is known as MAFIA [6] which stands for MAximal Frequent Itemset Algorithm. MAFIA uses a depth-first search and several pruning techniques in order to reduce the search space. The algorithm conceives items in the lattice as sets of Head and Tail. Head is the itemset that identifies a single node while possible extensions of it form the Tail. This is referred to as HUT or Head-Union-Tail. Using HUT, MAFIA, applies several techniques to prune the lattice.

A lexicographic subset tree [25, 26, 27] order \(\leq_{L}\) is preserved, meaning that if an item \(i\) occurs before an item \(j\) then we denote it as \(i \leq_{L} j\).

Figure 2.8 shows a complete subset lattice for four items.
Figure 2.8: A sample of a complete subset lattice for four items.

The root of the lattice is denoted as $null$ and each level $k$ of the lattice consists of all the $k$-itemsets. All itemsets as well as any children generated from them are ordered using lexicographical order.

The itemsets just right below and the ones above the cut, form the frequent itemsets; whereas any remaining itemsets, form the infrequent ones. In this case (see Figure 2.8) all itemsets are frequent besides $\{1, 2, 3, 4\}$ and $\{1, 3, 4\}$.

A depth-first search on itself is not substantially better than analogous breadth-first search in terms of realizing performance gains. This is due to the fact that the search space manipulated remains the same. In order to reduce the search space MAFIA employees a number of effective techniques [6].

The first technique is referred to as **Parent Equivalence Pruning (PEP)**. Consider operating on a node’s HUT that has $x$ as a head and $y$ as an item from its tail. If the transactions that contain $x$, also contain $y$ then it follows that any frequent itemset $I$ that holds $x$ but not $y$, has the frequent superset $I \cup y$. PEP removes item $y$ from the tail to the head of the node, as we are only interested in MFI's thus there is no need to
consider itemsets holding \( x \) but not \( y \). The algorithm for PEP is presented in Figure 2.9.

\[
\text{PEP}(\text{Current node } C, MFI) \{
\text{for each item } i \text{ in } C.\text{tail} \{
\text{newNode} = C \cup i
\text{if}(\text{newNode.supprot} = = C.\text{support})
\text{Move } i \text{ from } C.\text{tail to } C.\text{head}
\text{else if } \text{newNode is frequent}
\text{PEP}(\text{newNode}, MFI)
\}
\text{if}(C \text{ is a leaf and } C.\text{head is not in } MFI)
\text{Add } C.\text{head to } MFI
\}
\]

**Figure 2.9:** The Parent Equivalence Pruning method [6]

Another pruning technique used by MAFIA is named as *Frequent Head Union Tail (FHUT)*. A study performed by [27] shows that a node’s HUT is the largest possible frequent itemset that can be generated from that particular node and its subtree. Thus, in case a node’s HUT is detected as frequent, there is no need to further analyze any of its subsets, as it cannot produce any other frequent itemsets (it is important to note that we are only interested in MFIs). This technique results in pruning out the entire sub-tree rooted at that node. FHUT method is shown in Figure 2.10.

\[
\text{FHUT}(\text{node } C, MFI, \text{Boolean IsHUT}) \{
\text{for each item } i \text{ in } C.\text{tail} \{
\text{newNode} = C \cup i
\text{IsHUT} = \text{whether } i \text{ is the leftmost child in the tail}
\text{if}(\text{newNode is frequent})
\text{FHUT}(\text{newNode}, MFI, \text{IsHUT})
\}
\text{if}(C \text{ is a leaf and } C.\text{head is not in } MFI)
\text{Add } C.\text{head to } MFI
\text{if}(\text{IsHUT} \text{ and all extensions are frequent})
\text{Stop exploring this subtree and go back up tree to when IsHUT was changed to True}
\}
\]

**Figure 2.10:** The Frequent Head Union Tail method [6]
While FHUT uses direct counting of the support of an itemset in order to determine whether it is frequent or not, there is a more efficient technique named \textit{Hat Union Tail MFI (HUTMFI)} that merely checks whether the HUT is already included in the MFI list. If the test discovers HUT to be already included in the MFIs then the subtree rooted at that HUT’s node can be removed. The main advantage of HUTMFI is that it does not require exploration of any children while pruning, unlike FHUT which expands the entire leftmost branch of the tree. HUTMFI method is shown in Figure 2.11.

\begin{verbatim}
HUTMFI(Current node C, MFI) {
    name HUT = C.head \cup C.tail;
    if (HUT is in MFI)
        Stop searching and return
    for each item i in C.tail {
        newNode = C \cup I
        if (newNode is frequent)
            HUTMFI(newNode, MFI)
    }
    if (C.head is not in MFI)
        Add C.head to MFI
}
\end{verbatim}

\textbf{Figure 2.11}: The Head Union Tail MFI method [6].

MAFIA takes advantages of the three pruning techniques presented earlier in order to mine efficiently for maximal frequent itemsets. The complete MAFIA algorithm is show in Figure 2.12.
Figure 2.12: The complete MAFIA pseudo code [6].

2.5 Parallel Mining of Frequent Itemsets

Generally mining for frequent itemsets requires progressive knowledge collection and revision based on massive datasets. Achieving efficient parallel mining of frequent itemsets is a very challenging topic, as effective ways of partitioning these huge datasets across all nodes must be performed wisely; as well as balanced communication/computation ratio in order to reach global decisions that will not compromise the benefit trying to gain from parallelism. In the next chapter, parallel algorithms for mining frequent itemsets will be discussed extensively.
2.6 Summary

The idea about mining frequent itemsets is quite straightforward. You search within (usually large) datasets with the aim of discovering certain patterns that appear more times than a minimum threshold. These patterns are defined as frequent patterns. Association Rules are implication rules of the form $A \Rightarrow B$, meaning that the presence of an item $A$ in a transaction (purchase) implies the presence of another item $B$.

While the idea is straightforward the same does not apply for the way to achieve this. Several algorithms have been proposed to solve this problem. The first algorithm discussed in this paper was the Apriori algorithm that discovers frequent patterns via performing multiple passes over the database as well as candidate generation. FP-growth is another algorithm that discovers frequent patterns in a more efficient way that omits candidate generation and avoids multiple database scans. Finally MAFIA introduces a more competent idea that mines only for Maximal frequent itemsets and leads to more efficient operating speeds.
Chapter 3

Parallel Frequent Itemset Mining

3.1 Introduction

Typically the process of mining for Frequent Patterns is applied in large databases. For this reason parallel versions of several algorithms have been adapted, as the sequential versions tend to be very time consuming. Taking advantage of the computation power parallel machines provide, reduces significantly the time required to scan the dataset and generated Frequent Patterns [28, 29, 30, 31, 32]. Most of the existing parallel algorithms are based on sequential algorithms as plenty of state-of-art algorithms exist out there. Sequential versions of algorithms used for the process of frequent itemset mining appear to work sufficiently well for uni-processor architectures. As the bulk of the process emerges from the pattern generation stage, the parallel versions attempt to apply the computational power of parallel systems for distributing workload among multiple processors.

This section explains parallel versions of three Apriori-based algorithms namely Count Distribution, Data Distribution, Candidate Distribution, a parallel version of the FP- growth algorithm, and parallel MFI mining. The purpose of this chapter is to
examine how these algorithms can be implemented in parallel. Thus, no algorithmic
detail will be provided for algorithms discussed in previous sections.
For presentation purposes, assume $P_i$ to be a processor with id $i$.

### 3.2 The Count Distribution Algorithm

*Count Distribution* [30, 31], is a parallel Apriori-based algorithm used to mine for
frequent patterns. Each processor $P_i$ computes its local candidate itemset, along with
their support count, by performing a single pass over its local data partition at that
time. Information is maintained in a hash-table which is identical for each processor.
This procedure is accomplished by running a sequential Apriori on each processor.
All local counts are then accumulated and summed together to form a global support
count using a global reduction function [30]. This is illustrated in Figure 3.1. Global
reduction consists of two other operations. One of the operations is referred to as
*ReduceScatter* which is responsible for obtaining local support count communication
from a processor $P_i$, and the other operation is named *AllGather* operation which is
responsible for global support count communication.

Count Distribution seems to scale linearly to the number of records [30, 31] of the
dataset as all computations to find support counts can be done locally at each
processor having minor communication only at the end for accumulating the counts.
However, in case the hash-table structure cannot fit into the main memory of each
processor, it must be partitioned and support counts are computed by performing
multiple scans of the dataset (one scan for each partition of the hash-table). Thus the
algorithm is efficient only if candidate set generated leads to a hash-table with
reasonable size (relatively to main memory size). It is important to notice that the
number of candidates becomes larger as the number of distinct items in the dataset
increases or as the minimum threshold decreases. In general Count Distribution
works better for small number of distinct items and high levels of minimum
thresholds.
3.3 The Data Distribution Algorithm

The Count Distribution algorithm is attractive in the sense that no data movement is performed. All the counts are computed locally to each processor thus every processor can operate asynchronously on its own data. However this limits the ability of taking advantage of non-local memory parallel machines provide. The Data Distribution algorithm [30, 31] solves this problem by allowing each processor to compute the support counts of its locally stored subset of the candidate itemsets for all the transactions in the database.

In order for this to become feasible, the All-to-All broadcast is used, where each processor must scan its own partition of the data as well as other partitioned data located at remote processors. This results in every processor having to broadcast their data to all other participating processors as well as receive data from them.
Although this will solve the problem that Count Distribution carries with it, there are still negative effects as far as the burden placed in communication operations as there is a high communication overhead created due to data movement. Furthermore, such a communication scheme as this one causes the processors to become idle while waiting for data to be broadcasted resulting in wasting time that could have been manipulated for useful processing. The algorithm is illustrated in Figure 3.2.

![Figure 3.2: The Data Distribution Algorithm](image)

N: Number of transactions  M: size of candidate set  P: number of processors

3.4 The Candidate Distribution Algorithm

Both Count and Data distribution algorithms carry the limitation that there is some synchronization involved. Although in Count Distribution, each processor can compute its own candidates asynchronously, some synchronization is required when global counts are about to be summed. In case the workload is not perfectly balanced some processors may have to remain idle until others are finished. Similarly in Data
Distribution, synchronization is needed when data is broadcasted around the processors. Furthermore since any database transaction could support any candidate itemset, each transaction must be compared against the entire candidate set. For this reason Count Distribution needs to duplicate the dataset in every processor and Data Distribution needs to broadcast all of the transactions.

Candidate Distribution [31] combines the ideas used in both previous algorithms in order to overcome the problems associated with idle time, communication, and synchronization issues. This is achieved by duplicating the data on every processor’s local memory as well as partitioning the candidate set across processors. In this way every processor can proceed independently, using its part of candidates on its local data. There is no need to exchange data or counts using this algorithm. The only communication required is when pruning a local candidate set during the phase of pruning in candidate generation. However, there is no need for synchronization at this stage thus no processor has to remain idle until pruning updates from other processors arrive.

3.5 Parallel Mining of Frequent Itemsets without Candidate Generation

This section describes a parallel implementation of the serial FP-growth algorithm described in Chapter 2. As already mentioned the major bottleneck of Apriori-based algorithms arises from the fact that a large number of candidates may need to be generated. In addition multiple scanning of the database has a negative impact upon execution time.

The FP-growth algorithm adapts a different perspective, from Apriori-based ideas, by mining for frequent patterns avoiding the costly candidate generation phase. Also the use of a tree structure to store the transactions circumvents the need of multiple scans. Furthermore taking advantage of the aggregate memory and processing power of parallel machines, FP-growth appears to achieve higher speedups compared to its sequential version as well as scalable performance, especially for very large datasets.
The serial version of the algorithm has been described in detail in section 2.3, therefore attention will be given only to parallel issues concerning this idea.

The main idea is to have multiple FP-trees, one for each processor, allowing mining to be carried out in parallel until all the frequent patterns are generated [28, 29].

As in the serial version there are two stages to be followed: The Parallel Frequent Pattern Tree Construction, and the Parallel Mining and Frequent Itemset Generation.

Firstly, the dataset is divided equally among all participating processors $P_i$. Before the tree construction initiates, every processor must locate all local frequent 1-itemsets along with their local support count. The master processor gathers all local counts to be summed together and ultimately form a global support count to be broadcasted to all processors in the group.

Items having support count less than the minimum threshold are removed from the process. The next step is the FP-tree construction. Each processor constructs its own tree independently of others, based on the local dataset with respect to the global frequent items. The FP-tree construction on each individual processor follows the same procedure as the sequential version (see section 2.3). An example of construction of local FP-trees from local database partitions over 2 processors is shown in Figure 3.3.
Once the FP-trees are constructed the following step is to generate the conditional patterns based on the trees obtained. In order to do this, the header table is visited in a top-down manner and the local FP-tree is traversed in a bottom-up fashion, as in the sequential algorithm. Those patterns will facilitate the creation of the conditional FP-trees to be mined for frequent patterns. In order to construct the conditional FP-trees, all local conditional pattern bases for each frequent item must be accumulated from each processor [29]. Following this, each processor recursively traverses its assigned conditional FP-trees to discover frequent patterns in the presence of a particular item (as in the serial version). Figure 3.4 and Figure 3.5 in the next page, show the conditional pattern bases and conditional FP-trees for all frequent items assigned to the two processors.

Figure 3.3: Construction of local FP-trees from local database partitions over 2 processors ($P_1$, $P_2$)
Processor $P_0$ works on the conditional FP-trees for items 11, 13, and 14 while processor $P_1$ handles items 15 and 16. Item 12 has an empty conditional pattern base thus no further process in that item is required.
Since the algorithm performs only two scans of the database (in order to construct the FP-trees and work only with them) the disk Input/Output is almost negligible. The main communication and synchronization cost comes form the fact that all local conditional pattern bases need to be broadcasted among all processors [28, 29].

3.6 Parallel Mining of Maximal Frequent Itemsets

If we recall form Chapter 2, an itemset $X$ is a Maximal Frequent Itemset (MFI) if $X$ is frequent and there exists no superset of $X$ that is frequent.

At present we are unaware of any parallel implementation of the MAFIA algorithm presented in section 2.4, however we will discuss a different algorithm very similar to MAFIA.

This section discusses a parallel adaptation of an existing algorithm for mining maximal frequent itemsets known as Max-Miner [27, 32]. Max-Miner is very similar to MAFIA in terms of a look-ahead strategy that they both employ. Max-Miner refers to this pruning strategy as superset-frequency based pruning which is exactly the same as the FHAT pruning technique used in MAFIA. This look-ahead method essentially attempts to identify all frequent extensions of a particular node at an early stage, enabling the algorithm to stop processing that node any further (since we are only interested in MFIs). Unlike MAFIA, Max-Miner employs a breadth-first search in order to reduce database scans. The search space can be thought to be the same as in MAFIA as well as the HUT property. Furthermore Max-Miner uses a dynamic reordering scheme in order to increase the effectiveness of superset-frequency pruning. This scheme preserves an ascending ordering, based on support count of each item at each level, as items appearing in the tail of a node (remember the HUT) will also appear in most of the offspring nodes of that particular node. In effect, long patterns (MFIs) to be discovered at early stages [27].

Parallel Max-Miner (PMM) is an algorithm for mining MFIs and it is the parallel version of the sequential Max-Miner [32].
PMM divides the database in equal partitions based on the available processors having as a result each processor to work on the same number of transactions. The algorithm requires multiple passes over the database which will be discussed below. \( P_i \) is a processor with id \( i \), \( D_i \) is the local database partition for processor \( i \), \( F_k \) is the set of frequent \( k \)-itemsets, \( C_k \) includes all candidate groups generated based on \( F_{k-1} \) for pass \( k \), and \( GM \) is the set of all maximal frequent itemsets.

The algorithm works as follows [27]:

**Pass 1:**
- \( P_i \) counts the occurrences of each item in \( D_i \)
- \( P_i \) exchanges and merges the local counts with all other processors
- \( P_i \) determines \( F_1 \)

**Pass 2:**
- \( P_i \) generates \( C_2 \) by pairing the members of \( F_1 \)
- \( P_i \) counts the occurrences of the \( C_2 \) members in \( D_i \)
- \( P_i \) exchanges and merges the local counts of \( C_2 \) members with all other processors
- \( P_i \) determines \( F_2 \)
- \( P_i \) generates \( C_3 \)

**Pass \( k \), for \( k > 3 \):**
- \( P_i \) scans \( D_i \) to count the candidate groups in \( C_k \)
- \( P_i \) exchanges and merges the local counts of \( C_k \) members with all other processors
- \( P_i \) identifies frequent itemsets
- \( P_i \) inserts frequent itemsets into \( GM \), and keeps only maximal frequent itemsets in \( GM \)
- \( P_i \) generates \( C_{k+1} \) using \( F_k \) (superset-frequency pruning)

Although the communication and synchronization cost of Parallel Max-Miner does not seem to outperform significantly other Apriori-based algorithms, it is still
beneficial in the sense that Max-Miner avoids enumerating all potential frequent itemsets [27]. The number of database scans is still the same as Apriori.

A parallel version of the MAFIA algorithm would probably be able to achieve significant speed-ups compared to parallel Max-Miner as it performs additional pruning techniques as well as a different representation of the database which ultimately reduces the number of database scans. However this is beyond the scope of this project.

3.7 Summary

Serial algorithms are often poor in performing frequent itemset mining on very large datasets. Thus, several parallel algorithms have been proposed in order to overcome any hardware constrains and improve performance. Three algorithms have been described that are based on the Apriori method with Candidate distribution being the one that would provide a good balance between communication and computation among the tree. As in Chapter 2, two different types of ideas have been discussed. One is mining in parallel without candidate distribution that reduces the database scans and the expensive candidate generation. The other is the parallel version of ax-Miner, an algorithm that mines in for maximal frequent patterns. Although the communication and synchronization cost of Parallel Max-Miner does not seem to outperform significantly other Apriori-based algorithms it is still beneficial in the sense that Max-Miner avoids enumerating all potential frequent itemsets.
4.1 Introduction

There is work concerning outlier mining which detects data objects that are different or inconsistent with the remaining data. First of all, identifying outliers does not directly imply that all MSUs are detected and second, most techniques for that purpose apply only to continuous data and they are not suitable for datasets containing both continuous and categorical data [13].

[12] explains how mining of MFIs can be extended to discover MSUs; however not only the algorithm does not lead easily to the discovery of MSUs, but also it does not produce good candidate MSUs all of the times.

There is a great relationship between what the algorithm does which will be discussed later on in this paper, and an existing idea known as \( k \)-anonymity. We will get into more depth on this idea since our algorithm appears theoretically to be able to work as a preprocessing tool for this idea.

Moreover, a novel algorithm for mining infrequent patterns referred to as \( MINIT \) will also be discussed more extensively since it appears to be similar.
4.2 \( k \)-anonymity

4.2.1 Notation and Objectives

As stated earlier, a quasi-identifier is a combination of non-sensitive attributes that can be linked with external data and possibly re-identify individuals which are supposed to be undisclosed.

Some notation will facilitate in explaining how the idea of \( k \)-anonymity works in practice.

We denote a private table that is said to be in an anonymized form (i.e. sensitive attributes being removed) with \( PT \). Thus a quasi-identifier is a set of attributes in \( PT \) that it is possible to be linked to other external data and re-identify certain individuals. Given a table \( T(A_1, \ldots, A_n) \), a set of attributes \( \{ A_i, \ldots, A_j \} \subseteq \{ A_1, \ldots, A_n \} \), and a tuple \( t \in T \),

\( t_i[A_i, \ldots, A_j] \) indicates a chain of the values \( A_i, \ldots, A_j \) in \( t \) (i being the row associated with that tuple), \( T[A_i, \ldots, A_j] \) indicates the projection, including duplicates, of attributes \( A_i, \ldots, A_j \) in \( T \). Also, \( |T| \) indicates the number of tuples in \( T \).

The objective in this task is the releasing of information in a (still) useful form that will prevent anyone from extracting useful information from it resulting in disclosing individuals.

4.2.2 \( k \)-anonymity Principle

An idea introduced by [18] is referred to as \( k \)-anonymity which simply states that there should be no less than \( k \) occurrences of a quasi-identifier thus when \( k > 1 \) no entity is unique. The definition is more formally stated [14] as “Each release of data must be such that every combination of values of quasi-identifiers can be indistinctly matched to at least \( k \) individuals”. Thus if \( T(A_1, \ldots, A_n) \), is a table and \( Q_T \) is a quasi-identifier related to it, then in order for \( T \) to comply the \( k \)-anonymity requirement, the values in \( T[Q_T] \) must show up at least \( k \) tuples.
When tuples fail to satisfy the k-anonymity requirement, some masking techniques must be used in order to ensure protection. Masking is the idea of applying transformation methods in order to modify the initial data form. This includes methods that fall under two main categories: Perturbative and non-Perturbative [17]. What Perturbative methods do is that they remove unique combinations from the dataset and add new unique combinations in order to create confusion while maintaining statistical confidentiality.

On the other hand, non-Perturbative methods do not deform the dataset; instead they count on generalization and suppression techniques. We will get into more details on this in the next paragraph since this is the method used by k-anonymity.

The approach on providing k-anonymity relies on two masking techniques namely Generalization and Suppression [18].

### 4.2.3 Generalization and Suppression

Generalization is a way of creating new categories for the data that are more general (as the name implies) than the original ones [14, 18]. For example a more general form of an attribute that can take the values man and woman would be the value person. Another example is a ZIP code attribute that instead of providing the exact code i.e. 48091, we could expose a more general form like 4809* which states that this person lives in an area with ZIP code 48090 – 48099. It is possible to have several levels of generalization. For instance if the ZIP code 4809* is not enough to satisfy the requirements we can move to a more general level such as 480**. To visualize the process, let us consider a more concrete example of a generalized private table $PT$ that satisfies k-anonymity given $k = 2$ and $Q_T = \{\text{Race, Birth, Gender, ZIP}\}$ which is show in Figure 4.1.
Therefore as illustrated above, according to the requirement, each sequence of values in $PT[Q_T]$, appears at least in 2 occurrences of those values in $PT[Q_T]$.

In particular, $t_1[Q_T] = t_2[Q_T]$, $t_3[Q_T] = t_4[Q_T]$, $t_5[Q_T] = t_6[Q_T]$, $t_7[Q_T] = t_8[Q_T] = t_9[Q_T] = t_{10}[Q_T]$, and $t_{11}[Q_T] = t_{11}[Q_T]$.

Suppression is a more straightforward idea that completely removes parts of the dataset [14, 18]. It can be applied to individual attributes or even to entire tuples. Consider for instance a data release for a mortgage company [33]. The original table is shown in Figure 4.2. The anonymized table is shown in Figure 4.3.
### Chapter 4

**Related Work**

<table>
<thead>
<tr>
<th>Name</th>
<th>Gender</th>
<th>Marriage Status</th>
<th>Age</th>
<th>Sports Car</th>
<th>Loan Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>-</td>
<td>Young</td>
<td>-</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>-</td>
<td>Young</td>
<td>-</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>-</td>
<td>Young</td>
<td>-</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>-</td>
<td>Old</td>
<td>-</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>-</td>
<td>Old</td>
<td>-</td>
<td>Bad</td>
<td></td>
</tr>
<tr>
<td>Male</td>
<td>-</td>
<td>Old</td>
<td>-</td>
<td>Bad</td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>Bad</td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Bad</td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Bad</td>
<td></td>
</tr>
<tr>
<td>Female</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>Bad</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 4.3: Example of a dataset after suppression is applied*

The *Name* column has been completely removed from the set before suppression has taken place as it forms a part of a sensitive data and cannot be released. Partial suppression has been applied for the columns *Marriage Status* and *Age* and a total suppression for the column *Sports Car*. The resulting effect of this is achieving *k*-anonymity as in this way it is now impossible now to use any combination of quasi-identifiers and re-identify an individual.

#### 4.2.4 Minimal Generalization with Suppression

The major issue with both techniques is that they need to be used in moderation. Although the more generalized and suppressed the data become, the more protected it will be, the data may then be useless as the information left to use may be minimal. An effective method should be one that provides a good tradeoff between disclosure risk and information loss [17]. Ensuring that this is maintained, requires the use of minimal generalization and minimal suppression[14, 18].

In this context, the word *minimal* means that we should generalize or suppress only up to a level so *k*-anonymity is merely satisfied and no more than that. Suppression in general assists the generalization process especially when some outliers will cause generalization to be applied at high levels. Thus suppression leads to satisfaction of
It is clear that these two methods produce better results when they are applied together. Therefore we must find an optimal way to take advantage of them as the more generalized the data are, the lower the precision and the greater the suppression and the less complete the data will be. According to [14] it is better to suppress more records than to enforce more generalization. This is due to the fact that suppression affects only individual attributes or records, unlike generalization which affects all the values associated with a specific attribute.

Therefore the optimal process is to suppress up to a given maximum threshold \( \text{max\_sup} \), and then apply as much generalization as necessary (if necessary) in order to satisfy the \( k \)-anonymity requirement [18].

In order to illustrate the process consider the private table \( PT \) shown in Figure 4.4, where the \( k \)-anonymity condition is \( k = 2 \). While the dataset is being generalized in different levels, there are still some attribute combinations that are unique. Therefore the consequent \textit{minimal generalization with suppression} is presented in Figure 4.5.

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Asian} & 94138 \\
\text{Asian} & 94138 \\
\text{Asian} & 94142 \\
\text{Asian} & 94142 \\
\text{Black} & 94138 \\
\text{Black} & 94141 \\
\text{Black} & 94142 \\
\text{White} & 94138 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Person} & 94138 \\
\text{Person} & 94138 \\
\text{Person} & 94142 \\
\text{Person} & 94142 \\
\text{Person} & 94138 \\
\text{Person} & 94141 \\
\text{Person} & 94142 \\
\text{Person} & 94138 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Asian} & 9413* \\
\text{Asian} & 9413* \\
\text{Asian} & 9414* \\
\text{Asian} & 9414* \\
\text{Black} & 9413* \\
\text{Black} & 9414* \\
\text{Black} & 9414* \\
\text{White} & 9413* \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Person} & 9413* \\
\text{Person} & 9413* \\
\text{Person} & 9414* \\
\text{Person} & 9414* \\
\text{Person} & 9413* \\
\text{Person} & 9414* \\
\text{Person} & 9414* \\
\text{Person} & 9413* \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Person} & 9413** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Person} & 9413** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Person} & 9413** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Race} & \text{ZIP} \\
\hline
\text{Person} & 9413** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\text{Person} & 9414** \\
\text{Person} & 9414** \\
\text{Person} & 9413** \\
\hline
\end{array}
\]

**Figure 4.4:** Example of a Private Table \( PT \) and its generalized tables
Figure 4.5: Example of generalized tables, with suppression for the Private Table $PT$ of Figure 4.4

Whether a generalization is considered as minimal or not it depends on the maximum threshold given for suppression. From this example the following generalizations are considered to be minimal:

- $max\_sup = 0$: $GT_{[4]}$ ($GT_{[0]}$, $GT_{[1]}$, $GT_{[2]}$, and $GT_{[3]}$ suppress more rows than it is permitted, $GT_{[5]}$ is not minimal because of $GT_{[4]}$).
- $max\_sup = 1$: $GT_{[1]}$ and $GT_{[3]}$ ($GT_{[0]}$ and $GT_{[2]}$ suppress more rows than it is permitted, $GT_{[4]}$ is not minimal because of $GT_{[1]}$, and $GT_{[3]}$ is not minimal because of $GT_{[1]}$ and $GT_{[3]}$).
- $max\_sup = 2, 3$: $GT_{[1]}$ and $GT_{[2]}$ ($GT_{[0]}$ suppresses more rows than it is permitted, $GT_{[3]}$ is not minimal because of $GT_{[2]}$, $GT_{[4]}$ and $GT_{[5]}$ are not minimal because of $GT_{[1]}$ and $GT_{[3]}$).
- $max\_sup = 4$: $GT_{[0]}$ (all other generalized tables are not minimal because of $GT_{[0]}$).
4.2.5 *k*-anonymity Attacks

Sometimes, even when a dataset fulfills the *k*-anonymity requirement, there is still the possibility for someone to formulate an attack and re-identify individuals. Attacks to *k*-anonymity include [15]:

**Unsorted matching attack against *k*-anonymity:**
This is concerned with the order that the records appear in the table being released. For example, Figure 4.6, shows a private table *PT* and two possible generalizations of it. Making those two generalized tables public even for different purposes, you can very easily re-identify any record by matching the corresponding tuples from each table. This problem, though, can be easily avoided by re-ordering the tuples of the table.

<table>
<thead>
<tr>
<th>Race</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asian</td>
<td>02138</td>
</tr>
<tr>
<td>Asian</td>
<td>02139</td>
</tr>
<tr>
<td>Asian</td>
<td>02141</td>
</tr>
<tr>
<td>Asian</td>
<td>02142</td>
</tr>
<tr>
<td>Black</td>
<td>02138</td>
</tr>
<tr>
<td>Black</td>
<td>02139</td>
</tr>
<tr>
<td>Black</td>
<td>02141</td>
</tr>
<tr>
<td>Black</td>
<td>02142</td>
</tr>
<tr>
<td>White</td>
<td>02138</td>
</tr>
<tr>
<td>White</td>
<td>02139</td>
</tr>
<tr>
<td>White</td>
<td>02141</td>
</tr>
<tr>
<td>White</td>
<td>02142</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Race</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>02138</td>
</tr>
<tr>
<td>Person</td>
<td>02139</td>
</tr>
<tr>
<td>Person</td>
<td>02141</td>
</tr>
<tr>
<td>Person</td>
<td>02142</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Race</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asian</td>
<td>0213*</td>
</tr>
<tr>
<td>Asian</td>
<td>0214*</td>
</tr>
<tr>
<td>Asian</td>
<td>0214*</td>
</tr>
<tr>
<td>Asian</td>
<td>0214*</td>
</tr>
<tr>
<td>Black</td>
<td>0213*</td>
</tr>
<tr>
<td>Black</td>
<td>0214*</td>
</tr>
<tr>
<td>Black</td>
<td>0214*</td>
</tr>
<tr>
<td>Black</td>
<td>0214*</td>
</tr>
<tr>
<td>White</td>
<td>0213*</td>
</tr>
<tr>
<td>White</td>
<td>0213*</td>
</tr>
<tr>
<td>White</td>
<td>0214*</td>
</tr>
<tr>
<td>White</td>
<td>0214*</td>
</tr>
</tbody>
</table>

**Complementary release attack against *k*-anonymity:**
The previous example was based on a quasi-identifier that was built from the combination of all released columns. This, however, is not always the case. In many cases the quasi-identifier contains only a subset of the released columns. In this case any subsequent releases of tables related to the previous one, must be released with care to prevent linking back and obtain information which is not supposed to be
disclosed. For example Figure 4.7 illustrates a private table PT before masking takes place. Figure 4.8 shows a generalization, based on $QI_T = \{\text{Race, Birth, Gender, ZIP}\}$, that satisfies $k$-anonymity for $k = 2$.

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Race</th>
<th>Birth</th>
<th>Gender</th>
<th>ZIP</th>
<th>Medical Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>Black</td>
<td>09/20/1965</td>
<td>Male</td>
<td>02141</td>
<td>Short breath</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Black</td>
<td>02/14/1965</td>
<td>Male</td>
<td>02141</td>
<td>Chest pain</td>
</tr>
<tr>
<td>$t_3$</td>
<td>Black</td>
<td>10/23/1965</td>
<td>Female</td>
<td>02138</td>
<td>Painful eye</td>
</tr>
<tr>
<td>$t_4$</td>
<td>Black</td>
<td>08/24/1965</td>
<td>Female</td>
<td>02138</td>
<td>Wheezing</td>
</tr>
<tr>
<td>$t_5$</td>
<td>Black</td>
<td>11/07/1964</td>
<td>Female</td>
<td>02138</td>
<td>Obesity</td>
</tr>
<tr>
<td>$t_6$</td>
<td>Black</td>
<td>12/01/1964</td>
<td>Female</td>
<td>02138</td>
<td>Chest pain</td>
</tr>
<tr>
<td>$t_7$</td>
<td>White</td>
<td>10/23/1964</td>
<td>Male</td>
<td>02138</td>
<td>Short breath</td>
</tr>
<tr>
<td>$t_8$</td>
<td>White</td>
<td>03/15/1964</td>
<td>Female</td>
<td>02139</td>
<td>Hypertension</td>
</tr>
<tr>
<td>$t_9$</td>
<td>White</td>
<td>08/13/1965</td>
<td>Male</td>
<td>02139</td>
<td>Obesity</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>White</td>
<td>05/05/1964</td>
<td>Male</td>
<td>02139</td>
<td>Fever</td>
</tr>
<tr>
<td>$t_{11}$</td>
<td>White</td>
<td>02/13/1967</td>
<td>Male</td>
<td>02138</td>
<td>Vomiting</td>
</tr>
<tr>
<td>$t_{12}$</td>
<td>White</td>
<td>03/21/1967</td>
<td>Male</td>
<td>02138</td>
<td>Back pain</td>
</tr>
</tbody>
</table>

**Figure 4.7:** A private table PT before release

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Race</th>
<th>Birth</th>
<th>Gender</th>
<th>ZIP</th>
<th>Medical Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>Black</td>
<td>1965</td>
<td>Male</td>
<td>02141</td>
<td>Short breath</td>
</tr>
<tr>
<td>$t_2$</td>
<td>Black</td>
<td>1965</td>
<td>Male</td>
<td>02141</td>
<td>Chest pain</td>
</tr>
<tr>
<td>$t_3$</td>
<td>Person</td>
<td>1965</td>
<td>Female</td>
<td>0213*</td>
<td>Painful eye</td>
</tr>
<tr>
<td>$t_4$</td>
<td>Person</td>
<td>1965</td>
<td>Female</td>
<td>0213*</td>
<td>Wheezing</td>
</tr>
<tr>
<td>$t_5$</td>
<td>Black</td>
<td>1964</td>
<td>Female</td>
<td>02138</td>
<td>Obesity</td>
</tr>
<tr>
<td>$t_6$</td>
<td>Black</td>
<td>1964</td>
<td>Female</td>
<td>02138</td>
<td>Chest pain</td>
</tr>
<tr>
<td>$t_7$</td>
<td>White</td>
<td>1964</td>
<td>Male</td>
<td>0213*</td>
<td>Short breath</td>
</tr>
<tr>
<td>$t_8$</td>
<td>White</td>
<td>1964</td>
<td>Female</td>
<td>0213*</td>
<td>Hypertension</td>
</tr>
<tr>
<td>$t_9$</td>
<td>Person</td>
<td>1965</td>
<td>Male</td>
<td>0213*</td>
<td>Obesity</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>White</td>
<td>1964</td>
<td>Male</td>
<td>0213*</td>
<td>Fever</td>
</tr>
<tr>
<td>$t_{11}$</td>
<td>White</td>
<td>1967</td>
<td>Male</td>
<td>02138</td>
<td>Vomiting</td>
</tr>
<tr>
<td>$t_{12}$</td>
<td>White</td>
<td>1967</td>
<td>Male</td>
<td>02138</td>
<td>Back pain</td>
</tr>
</tbody>
</table>

**Figure 4.8:** $GT_{[1]}$ A generalized table for the private table in Figure 4.7

$GT_{[1]}$ works fine in terms of satisfying the requirements. However, consider now the subsequent release of another generalized table $GT_{[2]}$ using the same quasi-identifiers as before. $GT_{[2]}$ is shown in Figure 4.9.
The problem that arises in this case, is that even if tuples of $GT_{[2]}$ are sorted in a different order information can be used in combination with $GT_{[1]}$ and the linking table shown in Figure 4.10 can be generated.

Thus the $k$-anonymity is no longer satisfied as tuples $t_7$ and $t_8$ are now unique entities identifying individual records. The problem arises from the fact that $GT_{[2]}$ was based on the same quasi-identifier as $GT_{[1]}$. In order to overcome this problem, the optimal quasi-identifier must use another one such as $QI_{[GT_{[2]}]} = \{ QI_{[GT_{[2]}]} \cup \{ Medical\ Condition \} \}$. 
**Temporal attack against k-anonymity:**

Data is very frequently changed by insertions, deletions, and updates. Any change may affect a set of attributes or even entire tuples. Changing the data dynamically may have a negative impact on maintaining protection over time. Consider a private table $PT$ and a generalized release for it $GT_{[1]}$. After releasing $GT_{[1]}$ someone applies some changes to $PT$. By releasing now another generalized table $GT_{[2]}$ for $PT$ we allow the possibility of linking the two generalized tables and producing a linking table to $PT$. Therefore, each release of a generalized table must take into consideration any previous releases and any changed made over time in order to ensure protection.

**4.2.6 k-anonymity Weaknesses**

The attacks presented above demonstrate cases that can be avoided when care is taken. However $k$-anonymity does have drawbacks that have not yet been addressed. There is some cost that $k$-anonymity carries with it [17]. First there is high computational cost in determining the optimal generalization due to their exponential number. A second problem is that it requires choosing which categories will be used in order to generate the generalization.

The reason why this is a problem is because given labels will not match all kinds of uses for that data. Data is released for different purposes thus different labels will give a different meaning to each case.

These issues above are concerned with the algorithm itself, however there are some other problems presented by [16] which reveal cases in which $k$-anonymity fails to deliver the promised results.

One major problem is referred to as *Homogeneity attack* which involves a process by which “$k$-anonymity can create groups that leak information due to lack of diversity in the sensitive attribute” [16]. For instance if there exists a quasi-identifier with fields (age, zip code) and a sensitive field (medical condition), in case every person in the same age range of a certain area has the same medical condition you
can conclude that a person you know around that age, living in that area has this medical condition.

Another problem is *Background knowledge attack*. This is concerned with the fact that “$k$-anonymity does not protect against attacks on background knowledge” [16]. Based on our previous example let us assume that you know a Japanese person that was admitted to the hospital. Also let us assume that searching the data reveals that there is a group of people that match the same attribute as this person with a variation of two kinds of medical conditions: heart disease and viral infection. Knowing that Japan has a very small number of people suffering form heart disease, we can infer that the person we know suffers from viral infection.

### 4.3 Mining Minimal Infrequent Itemsets

So far Frequent Itemset Mining has attracted plenty of attention due to the importance of discovering association rules [5, 8, 45, 46]. It appears that much less attention has been given to Mining Infrequent Itemsets as the only work related to this domain is the algorithms MINIT [34] and SUDA2[4]. SUDA2, which was developed for detecting Minimal Unique Itemsets (itemsets with no unique proper subsets), is described extensively in a later section. At the moment we will concentrate on MINIT (**MIN**imal **IN**frequent **IT**emsets), which is the first algorithm designed specifically for mining *minimal infrequent itemsets (MIIs)*, and it is actually based on SUDA2.

An Infrequent Itemset is an itemset that has support count less than a minimum support threshold $\tau$. A Minimal Infrequent Itemset is an itemset that it is infrequent and has no infrequent proper subsets.

Introducing some notation used here will facilitate understanding the algorithm as we go through. Most of the ideas described here are similar to those used for mining frequent itemsets. Thus, we assume that there is a clear understanding of any previous related discussions.
Let $P = \{i_1, i_2, \ldots, i_n\}$ be a set of items. An itemset $I$, is a set of items such as $I \subseteq P$. A dataset, $D = \{t_1, t_2, \ldots, t_R\}$ is a set of records of the form of $t_i = (I_i, T_i)$, where $i$ identifies the record number and $T_i \subseteq P$. The support of an itemset, $I$, is denoted as: $D(I) = \{t_i \in D : I \subseteq T_i\}$.

The MINIT algorithm is presented in Figure 4.11.

---

**Input:**

- $D =$ input dataset with $N$ rows and $L$ columns of binary numbers
- $max_k =$ upper bound on cardinality of MII to find in the search
- $V[1:L] =$ a Boolean vector indicating viability of each item

**Output:**

- A listing of all MIIs for dataset $D$ with cardinality less than $max_k$

**Algorithm:**

compute $R \leftarrow$ list of all items in $D$ in ascending order of support

if $max_k = = 1$ /* stopping condition */ then

    return all items of $R$ that appear less than $\tau$ times in $T$

else

    $M \leftarrow \emptyset$

    for each item $i_j \in R$ do {

        $D_j \leftarrow D(\{i_j\})$

        $V[i_j] \leftarrow false$

        $C_j \leftarrow$ recursive call to MINIT($D_j, V, maxk -1$)

        for each candidate itemset $I \in C_j$ do {

            if $I \cup \{i_j\}$ is a MII in $D$ then

                $M \leftarrow M \cup (I \cup \{i_j\})$

            } //end if

        } //end for

    } //end for

    return $M$

} //end if

Figure 4.11: Pseudo code for MINIT($D,[1:L], maxk$) [34]

MINIT works as follows:

First a full scan of the dataset is performed in order to obtain the support count of each item. Then a list is created an every item is placed on that list in ascending
order of support count. MINIT is applied recursively on the dataset working on each item in the list starting from the one with the lowest support count, generating candidate MII which will eventually be checked against the original dataset. Only items with higher rank than the current working item are considered in the process.

One way to implement the idea of working only with items of higher rank is to use a vector that will keep track of whether an item is “alive” or not at a certain level during recursion.

As considerable amount of time is required to search through the dataset looking for transactions that include a particular item, a pre-process step is performed in which a linked-list structure is created for each item that holds pointers to all the transactions related to this item, sorted in ascending order. At each level of recursion a new sub-table is created and pre-processed, that corresponds to the support set $D(\{i_j\})$.

We will not discuss any properties used in this algorithm since the algorithm inherits the properties of SUDA2, which will be discussed in detail at a later chapter.

Although MINIT is very similar to SUDA2, there are important differences between them such as the input representation and the fact that MINIT investigates minimal infrequent itemsets whereas SUDA2 is seeking for minimal unique itemsets.

### 4.4 Summary

It appears that there exists only a limited amount of work relating to our problem area. $k$-anonymity seems quite efficient on providing statistical disclosure control; however it also has its shortcomings. MINIT, the first algorithm that searches for infrequent patterns, uses some clever properties thus the results are guaranteed to be fast and efficient. The main problem with these ideas is that none of them focuses on Minimal Sample Uniques detection which are the itemsets we are interested in. Once these itemsets have been detected, the problem of disclosing information becomes a “child's play”.

60
Chapter 5

Sequential Program Design

5.1 Depth-First Search Vs Breadth-First Search

*Depth-first Search (DFS)* [35] is an algorithm used for traversing a tree structure or a graph. It employs the idea of starting the search from the root (the only node that has no parent node) and moving down the graph going as deep as possible until the target is discovered, or until it reaches a leaf node (a node with no children) in which case it backtracks returning to the most recent node that has children that haven’t been searched yet. Consider for instance the tree structure shown in Figure 5.1.

```
               A
             /   \
            B     C
          / \    /   \
         D   E  F    G
         \   \  \    \  
          \   \  \    \\  
           H   I
```

*Figure 5.1:* A tree structure consisting of nine nodes
A DFS will start traversal from node \( A \). Then it will go down to node \( B \), and then to node \( D \). Assuming that the target is not found yet, the algorithm, backtracks from node \( D \) to node \( B \) in order to explore any children that haven’t been explored yet. Similarly, after visiting node \( E \) it will move to node \( F \) and then it will backtrack. Thus the order in which nodes are visited will be: \( A, B, D, E, F, G, H, I, C \).

Another searching algorithm is known as *Breadth-first search (BDS)* [35]. Like DFS, BDS starts the traversal from the root of the structure. However, unlike DFS, BDS does not descend down the graph to the deepest possible level (at any given branch). Instead, it explores all neighboring nodes at a given level while moving down the graph. Therefore the sequence in which the nodes of the tree shown in Figure 5.1 will be traversed is the following: \( A, B, C, D, E, F, G, H, I \).

Although both search algorithm are similar, DFS is sometimes preferred to BDS, especially when searching tree structures, as it operates at a lower space complexity (it requires less memory) due to the fact that only nodes of the current path need to be recorded [35].

### 5.2 Mining for Minimal Unique Itemsets in Microdata

We can think of Minimal Unique Itemsets (MUI) as a subset of infrequent items. Mining for Infrequent Patterns (as seen in the previous chapter) can be thought of as the opposite of mining for frequent patterns. That is, we search a dataset to discover patterns that appear less than a specified threshold. A MUI is an itemset that appears in only one row of the dataset and has no unique proper subsets. We refer to these itemsets as Minimal Sample Unique (MSUs) itemsets. We indicate the number of items that occur in an itemset with the letter \( k \). An MSU with \( k \) items is denoted as \( k \)-MSU. As stated in [12] the Minimal Uniques problem is as follows: “given a set of microdata containing \( C \) columns and \( R \) rows, find all \( k \)-MSU, \( 1 < k < \text{max}k \), where \( \text{max}k < C \) and is user-specified.”
Microdata are usually composed of equal-length rows’ consisting of continuous as well as categorical attributes. They are generally small in size with the number of rarely exceeding a few hundreds.

It is important to notice that there are some differences between frequent and infrequent pattern mining. For instance, an itemset here is not the same as an itemset in frequent pattern mining.

Here we consider an itemset as a set of items with no two items corresponding to the same column whereas in frequent pattern mining no column constraints apply. Furthermore in frequent pattern mining every subset of a frequent itemset must itself be frequent whereas here this is not true. Therefore it is not as easy to just adopt searching techniques used in frequent pattern mining.

### 5.3 SUDA – Special Unique Detection Algorithm

The Special Unique Detection Algorithm is a sequential algorithm [3] that is able to locate and grade individual records with high risk of disclosure. A record can contain more than one unique pattern. The risk is measured according to the number of patterns and size of each pattern in each record. The more unique patterns in a record and the smaller they are in size, the riskier this record is. Those unique patterns are the MSUs we have seen in section 1.1. The algorithm operates using a depth-first search to traverse the search space while repeatedly sorts the data using a variation of the bucket sort [3]. The bucket sort simply distributes the data in chunks to a number of “buckets” and then using a different sorting algorithm (such as insertion sort), sorts the data inside each bucket in isolation. After that, the data is returned to its original array; however, this time in order.

SUDA is designed around the observation that “Every superset of a unique attribute set (minimal or not) is itself unique” [3]. This is known as the Superset Relationship. This idea was inspired by the Apriori property mentioned in section 2.2. SUDA incorporates a bottom-up approach thus taking advantage of the Superset

---

1 a set of attributes that can be linked directly with external known data to identify individuals
Relationship meaning that once a set of attributes is found to be unique there is no need to further check any of its supersets since they cannot produce a minimal unique pattern. This idea is applied in order to reduce the search space in the following way: all attribute sets that share the same prefix are generated in succession thus in cases where an attribute set is found to be unique, any superset of this unique set can be ignored for that record.

For instance consider a dataset containing the attributes A, B, C, D, E, and F. By applying SUDA, we generate combinations of these attributes such as: A, AB, ABC, ABCD, ABCE, ABCF, ABD, ABDE, ABDF, ABE, ABF, AC, ACD etc …

Now, consider attribute sets having prefix ABC (i.e. ABC, ABCD, ABCE, ABCF). If ABC is identified as a MSU at a particular record, then all supersets that have ABC as a prefix can be ignored for that record as they cannot lead to more MSU’s. Since all those supersets appear after ABC, there is no need for extra retrieving of stored information. The process can just continue with different attribute sets.

Notice here the relationship between this technique and the pruning technique FHUT used in MAFIA mentioned in section 2.4. Using this pruning method, SUDA achieves significant reduction in the number of rows that need to be considered for each attribute set as well as minimizes memory usage [3, 4].

Furthermore the algorithm partitions the data using some grouping procedures which cluster together records that possess identical values for every attribute set generated.

Thus each cluster contains rows associated with a specific attribute subset. Any cluster that contains only one row indicates a unique itemset. This itemset can be checked immediately whether it is a minimal unique by simply ensuring that all subsets (of size $k-1$) of it are not unique.

This has the effect of minimizing the data storage needed to identify MSUs by localizing records needed for a given search. The final output of this algorithm is a count of the MSUs (up to a maximum user-specified size $M$) found for each record.

Figure 5.2 illustrates a dataset consisting of six rows and five columns. An item in this case is represented by taking a column label juxtaposed with a reference number.
associated with that column (i.e. \( C_3 \) means \( C = 3 \)). An itemset \( I \) is a collection of such items (i.e. \( I = \{ A_1, B_4, D_2 \} \)) having no items to correspond to the same column.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Row 2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Row 3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Row 4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Row 5</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Row 6</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 5.2:** Example of a dataset

### 5.3.1 Example

In order to gain a better understanding of how the algorithm works consider the tables shown in Figure 5.3 (shown in the next page). This is a case where the SUDA discovers the MSUs by processing the column subsets \( A, AB, ABC, ABCD, \) and \( ABCDE \).

The dataset is first sorted and partitioned according to column \( A \). So far no unique itemsets can be identified. Then each of these partitions is sorted independently (like bucket sort) on column \( B \) yielding three MSUs: Row 4: \( \{ A_2, B_4 \} \), Row 5: \( \{ A_1, B_3 \} \), and Row6: \( \{ A_2, B_3 \} \). We could recognize immediately that these were MSUs as they are all of size 2. Any size 1 MSU would have been removed from the first scan of the dataset. Further sorting on column \( C \), the algorithm, discovers Row 3: \( \{ A_1, B_4, C_2 \} \) to be unique. However it is not minimal since a subset of it: \( \{ A_1, C_2 \} \) has already been discovered as unique. Sorting continues on column \( D \) where two rows are identified as unique: Row 1: \( \{ A_1, B_4, C_1, D_2 \} \) and Row2: \( \{ A_1, B_4, C_1, D_1 \} \). Out of these two itemsets only the one at Row 2 is a MSU as minimal unique subsets of the itemset at Row 1 have already been discovered. The checking for \( ABCDE \) has now finished. The algorithm proceeds to traverse \( ABCE, ABD, ABDE, \) etc. The process carries on until all possible subsets have been checked or until a (user specified) maximum limit of size \( k \) (-itemset) has been reached.
Figure 5.3: The sorting process that SUDA accommodates

Row 1: 1 4 1 2 2
Row 2: 1 4 1 1 2
Row 3: 1 4 2 2 2
Row 4: 1 4 1 1 2
Row 5: 1 3 1 2 3
Row 6: 1 3 2 1 3

Row 1: 1 4 1 2 2
Row 2: 1 4 1 1 2
Row 3: 1 4 2 2 2
Row 4: 1 4 1 1 2
Row 5: 1 3 1 2 3
Row 6: 1 3 2 1 3

Row 1: 1 4 1 2 2
Row 2: 1 4 1 1 2
Row 3: 1 4 2 2 2
Row 4: 1 4 1 1 2
Row 5: 1 3 1 2 3
Row 6: 1 3 2 1 3

Row 1: 1 4 1 2 2
Row 2: 1 4 1 1 2
Row 3: 1 4 2 2 2
Row 4: 1 4 1 1 2
Row 5: 1 3 1 2 3
Row 6: 1 3 2 1 3

Row 1: 1 4 1 2 2
Row 2: 1 4 1 1 2
Row 3: 1 4 2 2 2
Row 4: 1 4 1 1 2
Row 5: 1 3 1 2 3
Row 6: 1 3 2 1 3

Row 1: 1 4 1 2 2
Row 2: 1 4 1 1 2
Row 3: 1 4 2 2 2
Row 4: 1 4 1 1 2
Row 5: 1 3 1 2 3
Row 6: 1 3 2 1 3

Figure 5.3: The sorting process that SUDA accommodates
5.4 A new recursive approach - SUDA2

Although SUDA is effective in identifying and grading MSUs, it is restricted only to a small number of columns [12]. The pruning technique, that SUDA incorporates, has an effect on reducing the number of records considered to be associated with each column subset. However, it is not as effective in reducing the numbers of column subsets considered [4]. For example in case any two rows are the same then all subset combinations will be generated for testing. SUDA2 not only employs an improved representation of the search space but also identifies a set of properties associated with MSUs that are used to design new pruning methods.

5.4.1 Search Space Representation

The search space is visualized as all possible different subsets of attributes in any row. SUDA2 reduces the search space every time an MSU is discovered. What this means is that whenever a new MSU is identified we can reduce the search space associated with this row in a way that this MSU is no longer a subset of this row’s attribute sets.

Consider again the dataset of Figure 5.2. The process of pruning the search space for this dataset is illustrated in Figure 5.4 (shown in the next page). The initial search space is illustrated at first followed by the resulting search space after removing 2-MSUs related with item $A_2$. The next table illustrates the remaining search space after removing 2-MSUs related with items $A_2$, $B_3$, $C_2$, and $D_1$. Finally the completed search for MSUs is illustrated in the last table. It is clear that the more MSUs are detected, the smaller the search space becomes [4].
**Figure 5.4:** The process of searching for MSUs
5.4.2 Properties of MSUs

Four properties of MSUs have been identified by [12]. These are **Support Row Property**, **Uniform Support Property**, **Recursive Property** and **Perfect Correlation Property**.

**Support Row Property:**
For a given \( k \)-MSU \( M \) at row \( R \) there must be at least \( k \) rows besides \( R \) that have itemsets different from \( M \), by exactly one item so \((k-1)\)-subset of \( M \) is matched [4]. These rows are known as the “support rows”.

**Uniform Support Property:**
A unique \( k \)-itemset that has an item \( X \) in each of its support rows cannot be an MSU since it would be a superset of the 1-MSU formed by that item. This idea is similar to the **Parent Equivalence Pruning (PEP)** technique, presented in section 2.4, which is adopted by the MAFIA algorithm.

**Recursive Property:**
Consider a \( k \)-MSU \( M \) in a row of table \( T \). Assume \( N \) is a \((k-1)\)-itemset resulting from removing an item \( I \) from \( M \). Let \( T_i \) be a subset of \( T \) containing only rows that \( I \) appears. It is clear that \( N \) is a \((k-1)\)-MSU in \( T_i \). If \( N \) was not minimal unique in \( T_i \) then \( M \) would not be unique in \( T \).

**Perfect Correlation Property:**
If two items \( I \) and \( J \) appear in exactly the same set of rows, they are referred to as perfectly correlated items. For an MSU \( M \) containing the item \( I \) there exist a corresponding MSU \( N \) with the item \( I \) replaced by item \( J \).

Following these new properties, along with the new representation of the search space, the algorithm works as follows:

5.4.3 Algorithmic Details

The input dataset is scanned and a list named **ITEM-LIST** is produced according to the repetition counts of all items that appear at least once. Each item in the list is
referred to as an anchor. These items are placed in the ITEM-LIST in ascending order of repetition count. If any 1-MSUs are found, they are recorded, and not included in the list as they cannot produce any further MSUs. This ordering facilitates in efficient pruning using the Uniform Support Property. The relative position within the list is known as rank.

Every MSU has an item with the lowest rank, named the reference item. We check for pairs of items that satisfy perfect correlation and one item of every pair is removed from the list. A table $T$ is created having each item replaced by its rank in regard to the ITEM-LIST. Then each item is associated with a set of rows that contain this item. This allows the set of rows in which an item occurs to be found right away.

The items in ITEM-LIST are regarded as independent sub-searches in ascending order of rank. A sub-table $T_i$ is created for each anchor which is made up of rows containing that anchor. SUDA2 is applied recursively to a sub-table in order to discover all $(k-1)$-MSUs $N$ in $T_i$, where $2 < k < C$, to be used as candidates for finding $k$-MSUs in table $T$. A candidate $(k-1)$-MSU, $N$, leads to a $k$-MSU only in case all items in $N$ have a rank higher than the anchor of that specific sub-table, and there exist a row in $T$ (and not in $T_i$) that holds all items in $N$ but not holding the reference item. We refer to this row as the special row. As soon as a MSU is identified, any corresponding MSUs that satisfy the perfect correlation property are added.

The pseudo code for SUDA2 is depicted in Figure 5.5 shown on the next page.
**Input:**

\[ T = \text{input dataset with } n \text{ rows and } m \text{ columns of integers} \]
\[ maxk = \text{maximum size of MSU in the search} \]

**Output:**

A set of MSUs for dataset \( T \)

**Algorithm:**

1. compute \( R \leftarrow \text{list of all items in } T \)
2. if \( maxk = 1 \) then
   - return all items of \( R \) that appear only once in \( T \)
3. else
   - \( M \leftarrow \emptyset \)
   - for each item \( i \in R \) do
     - \( T_i \leftarrow \text{subset of } T \text{ consisting of only rows of } T \text{ holding item } i \)
     - \( C_i \leftarrow \text{recursive call to SUDA2}(T_i, maxk -1) \)
     - for each candidate itemset \( I \in C_i \) do
       - if \( I \cup \{i\} \) is a MSU in \( T \) then
         - \( M \leftarrow M \cup (I \cup \{i\}) \)
       - end if
     - end for
   - return \( M \)
   - end if

---

**Figure 5.5:** Pseudo code for SUDA2 [4]

---

### 5.4.4 Example

Due to the complexity of the algorithm it is not always straightforward to perceive what exactly the program is doing. The use of an example that applies SUDA2 to the dataset \( T \) shown in Figure 5.6, may facilitate a better understanding of this algorithm.
Following the explanation given before, the dataset is first scanned, in order to acquire information about the support count of each item. The items are then sorted in ascending order based on their support count. Consequently, the ITEM-LIST is then formed by these items:

\[ \text{ITEM-LIST} = \{ A_2, B_3, C_2, D_1, E_2, E_3, A_1, B_4, C_1, D_2 \} \]

**Note:** For demonstration purposes this example differs slightly from reality in the following ways:

- Items in the ITEM-LIST are not replaced by their ranks
- The initial table is denoted with \( T \). Any sub-tables containing rows of \( T \) that item \( X \) occurs, are symbolized as \( T_x \), and any sub-tables of \( T_x \) containing rows that item \( Y \) occurs are indicated with \( T_{xy} \)
- No unique items are present
- There are no items in the dataset that satisfy the perfect correlation property
- No item occurs in all rows of the dataset

Starting with the first item of the list, \( A_2 \), a sub-table \( T_{A2} \), holding all the rows that this item occurs is created and shown in Figure 5.7.
The columns $A$ and $E$ are discarded as being redundant (appears in both rows) by the Uniform Support Property.

Six candidate MSUs can be seen immediately in table $T_{A2}$:
Row 4: $\{A_2, B_4\}$, $\{A_2, C_1\}$, $\{A_2, D_2\}$
Row 6: $\{A_2, B_3\}$, $\{A_2, C_2\}$, $\{A_2, D_1\}$
There is not need to check for the special row in this case as all candidate MSUs are of size 2.
The algorithm processes items $B_3$, $C_2$, and $D_1$ following the same approach as all of them have support count 2. Items with lower ranks are ignored.
Item $E_2$ with support count 3 is examined next and the sub-table $T_{E2}$ shown in Figure 5.8 is created to hold all the rows that contain that item.

![Figure 5.8: Sub-table $T_{E2C1}$ is extracted from $T_{E2}$](image)

Empty cells represent items with lower rank in the list. Columns $A$, $B$, and $E$ are discarded as redundant for this sub-table. No unique items appear to be present in $T_{E2}$ thus the next item in the ITEM-LIST, $C_1$, is used to create the sub-table $T_{E2C1}$ (also shown in Figure 5.8) which hold the rows that include item $C_1$ within the rows of $T_{E2}$.
Item $D_2$ appears to be unique in $T_{E2C1}$ thus $\{C_1D_2\}$ is a candidate MSU for $T_{E2}$. Item $D_2$ occurs independently of item $C_1$ (at row 3) in $T_{E2}$ meaning that the special row condition has been satisfied and thus $\{C_1D_2\}$ is actually a MSU in $T_{E2}$. Consequently the itemset $\{C_1D_2E_2\}$ is a candidate MSU in $T$. Once more the special row condition is satisfied as itemset $\{C_1D_2\}$ occurs independently of $E_2$ (at row 4 or row 5) in $T$. 

73
Similarly the sub-table $T_{E3}$ shown in Figure 5.9 is generated.

![Figure 5.9: Sub-table $T_{E3C1}$ is extracted from $T_{E3}$](image)

Two MSUs are distinguished right away: $\{A_1E_3\}$ and $\{B_4E_3\}$. Extracting sub-table $T_{E3C1}$ yields no unique items (it is empty). Consequently the search proceeds to the next item in the list which is $A_1$. The sub-table $T_{A1}$ is generated and shown in Figure 5.10.

![Figure 5.10: Analyzing and breakdown process for sub-table $T_{A1}$](image)

No unique items appear to be present in $T_{A1}$ therefore the sub-table $T_{B4A1}$ is generated from $T_{A1}$ which still shows no unique items. Extracting sub-table $T_{A1B4C1}$ from $T_{B4A1}$ reveals the itemset $\{C_1D_2\}$ to be a candidate MSU for $T_{B4A1}$. Item $D_2$ occurs independently of $C_1$ (at row 3) and therefore by the special row property $\{C_1D_2\}$ is a MSU in $T_{B4A1}$. 

74
Consequently \( \{B_4C_1D_2\} \) is candidate MSU for \( T_{A_1} \). \( \{C_1D_2\} \) occurs independently of \( B_4 \) (at row 5) and thus \( \{B_4C_1D_2\} \) is a MSU for \( T_{A_1} \). Now, \( \{A_1B_4C_1D_2\} \) is a candidate MSU for \( T \), which is verified by discovering that \( \{B_4C_1D_2\} \) occurs independently of \( A_1 \) (at row 4) in the entire dataset.

Finally items \( B_4, C_1, \) and \( D_2 \) yield no MSUs and therefore the search process is terminated.

### 5.4.5 Performance Accomplishments

SUDA2 has been compared with SUDA using real-world and synthetic datasets for the experiments. The results show that SUDA2 outperforms SUDA by several orders of magnitude, enabling datasets of considerable large number of columns to be tackled [4, 12].

### 5.5 Summary

This chapter discusses the sequential program design. Existing techniques used in mining of frequent itemsets were manipulated and used to create the initial algorithm, SUDA. Although the first algorithm was very effective at locating MSUs, there was still some room for improvement in terms of performance. This was the driving force behind the design of the second version, SUDA2. SUDA2 identifies additional recursive and non-recursive properties as well as new ways to take advantage of the search space, resulting in a solution that is superior in several orders of magnitude compared to its ancestor. The implementation of SUDA2 was reused for the implementation of a parallel algorithm, as it will be shown in the next chapter.
Chapter 6

Parallel Program Design

6.1 Parallel Computing and Parallel Programming

The techniques mentioned in the previous sections are usually process-intensive due to the fact that they usually have to deal with large datasets. One possible way to resolve this problem is to exploit as much parallelism as possible.

Parallel computing is defined as the simultaneous use of multiple compute resources to solve a computational problem [1]. Compute resources include either a single computer with multiple processors or an arbitrary number of computers connected via a network. A computational problem must be able to be broken down into discrete parts that can be computed simultaneously.

The major motivation behind parallel computing is the need to solve problems in less time than traditional serial computing. Additional reasons include the ability to overcome memory constrains and take advantage of multiple resources.

Parallel programming involves 4 steps: Decomposing computation into tasks, Assigning task to processes, Orchestrating necessary data access, communication and synchronization between processes, and finally mapping/binding of processes to processors [21]. The two most important parallel programming paradigms are shared-address-space and message-passing [19].
In the *shared-address-space* model there is a common address space shared between processes. This has the advantage of data sharing being very fast due to the proximity of memory to the processors. As processes may interfere with each other, various lock mechanisms may be required in order to ensure mutual exclusion in accessing shared memory, which adds a burden to the programmer. Furthermore this scheme lacks of scalability as an increase in processor number can raise geometrically the traffic on shared memory [2].

In the *message-passing* model there is no sense of global address space. Each process has its one private local memory. Processes can communicate with each other via passing messages. Data transfer involves cooperative primitives to be executed by each process (a send primitive must have a corresponding receive). In order to program using message-passing certain libraries must be used to manage data exchange between processors [2]. Unlike shared-address-space, the message-passing model offers scalability between processors and memory. An increase in processors will proportionally increase the memory size as well.

In this project the message-passing model will be used as the nature of our data does not manipulate the advantage of shared-address-space of having shared memory (as we deal with microdata). On the other hand message-passing offers great scalability and fast local memory access. This is ideal for our situation as datasets can be multiplied across each processor allowing faster accessing times for each unit. The system used in this project consists of a Hybrid parallel architecture where multiple SMPs are connected using a network interconnect. An SMP is a machine that consists of multiple identical processors using the same shared main memory. The next chapter provides more detail concerning the system used in this dissertation.

Furthermore, for our implementation, the Message Passing Interface (MPI) has been chosen. Further details on MPI along with the reasons that led to this choice are discussed in the relevant section.
6.2 The Message Passing Paradigm

As mentioned earlier, processes employed under the *message-passing* paradigm depend on exchanging messages to enable communication between them.

There are three message-passing modes: *Synchronous Message Passing, Blocking Send/Receive*, and *Non-blocking Send/Receive*.

In Synchronous Message Passing mode, when a process P executes a synchronous “send M to process Q”, it has to wait until process Q executes a corresponding synchronous “receive S from P”. Both processes will not complete their respective send/receive until message M is both sent and received.

In Blocking Send/Receive mode when a send is executed it does not wait for a corresponding receive. However it will not return until the message send is completed. When a ‘receive’ is executed it does not return until the message receipt is completed.

In Non-blocking Send/Receive mode when a message is sent it does not wait for a corresponding receive. In addition when a send/receive is executed they can return immediately after that send/receive, i.e. they do not wait for completion of the respective send/receive.

The first mode is classified as synchronous and the remaining two as asynchronous. The advantage with Blocking asynchronous is that it reduces waiting time. However additional buffer space is required in order to store (queue) the messages. Non-Blocking asynchronous can reduce waiting time to minimal but may cause problems by having messages arriving out of order.

In order to provide a solution to this problem a “status checking” or “wait-for” function is provided which forces process to wait until it is safe to continue.

The programmer may explicitly allocate sufficient buffer space required, otherwise the system may fail, or the parallel program may produce deadlocks.

The main motivation for using non-blocking mode is to overlap communication with computation.
6.3 Message Passing Interface – MPI

MPI is a standard specification for a library of message-passing functions [20]. It specifies this library in a language-independent form, and provides C, C++ and FORTRAN bindings.

The main reason for selecting MPI over other public-domain message-passing systems (such as PVM) is that MPI is expected to run faster within a large multiprocessor environment whereas other systems, like PVM, may work more effectively in a heterogeneous environment [23].

MPI library provides more than 125 functions of which six are the basic ones that can be applied to produce the simplest message-passing program. These functions are summarized in Figure 6.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes MPI</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminates MPI</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Determines the number of processes</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Determines the label of the calling process</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends a message</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives a message</td>
</tr>
</tbody>
</table>

**Figure 6.1:** A summary of the minimal-set of MPI functions

MPI_Init is used to initialize the MPI parallel environment. MPI_Finalize is to terminate the parallel environment. Both functions must be called by all processes in order to avoid unpredictable behavior.

A communication domain is formed by all processes that can communicate with each other. Variables of type MPI_Comm, called communicators, provide information about processes in the communication domain. A process may belong to many different communication domains. As all processes may need to communicate with each other, MPI uses a default communicator called MPI_COMM_WORLD that contains all processes involved in the parallel execution. MPI_Comm_size and MPI_Comm_rank are used to determine the size of the processes that belong to a communicator, and the rank of each process respectively.
Processes communicate with each other using \texttt{MPI\_Send} and \texttt{MPI\_Recv} in order to send or receive messages. The two functions are as follows:

\begin{verbatim}
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
            int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
\end{verbatim}

\texttt{MPI\_Send}'s arguments are: the buffer that stores the data to be sent, the number of entries in the buffer, the data type of the entries in the buffer, the destination of the message, and the type of message. \texttt{MPI\_Recv}'s arguments are: the buffer of the received message, the number of entries in the buffer, the data type of the entries in the buffer, the source of the message, and the type of message. An additional argument, called \texttt{status}, is provided in order to obtain information about the \texttt{MPI\_Recv} operation after the message has been received.

\section{Parallel SUDA2}

\subsection{Parallel design issues}

The first step to follow in writing a parallel program, is to break down the computation in tasks. The \textit{Granularity} of decomposition is determined by the number and size of tasks. Having a small number of large tasks is known as \textit{coarse-grained} and having a large number of small tasks is known as \textit{fine-grained}. A practical indicator for the performance of a parallel program is called the \textit{average degree of concurrency}. This specifies the average number of tasks that can run simultaneously during the entire execution of the program. \textit{The average degree of concurrency usually increases as the granularity of tasks becomes smaller (finer)} [19].
There two main techniques used in order to decompose a problem into tasks, namely *recursive decomposition* and *data-decomposition*.

Recursive decomposition is a technique that is used to induce concurrency in problems that can be solved using the divide-and-conquer strategy. Using this method the problem is divided into a set of independent sub-problems (usually of the same type). Then, recursion is applied to each of the sub-problems using the same idea of division resulting into smaller sub-problems. The same procedure continues until each of these sub-problems is simple enough to be solved directly. Each solution is then combined to provide a solution of the main problem. This idea results in natural concurrency since all of these sub-problem are autonomous and thus can be solved concurrently.

Data decomposition is a method for deriving concurrency in algorithms that operate on large data structures. Using this method, the data on which the computations are performed is first partitioned, and then it is used to induce a partitioning of the computations into (usually similar) tasks.

There are three ways of addressing the problem of partitioning the data. These are *partition on output data*, *partition on input data*, and *partition on both input and output data*.

Partitioning on output data involves decomposing the problem into tasks by partitioning the output in different parts, having each task computing its own part. The disadvantage of this, is that it can only be performed if each output is a natural function of the input. In cases where it is not, partitioning on input data can be used.

Partitioning on input data applies a similar idea to partitioning on output data. The main difference between them is that partitioning on input concerns the input data.

Finally, partitioning on both input and output data involves dividing both input and output (if possible) into different parts and distributing all possible combinations of them to tasks. Each task performs its local computations and then some pair-wise addition is used at the end.

SUDA2 is a good candidate for parallelism as its searches can be divided according to their rank [4, 12, 36].
A parallel version of SUDA2 was introduced by [36]. Our implementation in this project applies some optimization techniques in order to increase the speed-up and efficiency levels achieved so far.

The recursive decomposition technique can be naturally adapted for use in SUDA2 due to the recursive nature the algorithm utilizes. That is, recursively break down the problem into smaller sub-problems until the problem is small enough to be solved in a straightforward manner. In this case the main search can be broken into smaller sub-searches that can be assigned to different processors, solved simultaneously, and ultimately increase the speed-up that the algorithm operates.

### 6.4.2 SUDA2 – A Divide-and-Conquer Algorithm

Algorithms of the type of nature such as SUDA2 utilizes are sometimes referred to as divide-and-conquer algorithms. The main issues concerning these types of algorithms include: the branching factor, balance, data-dependence, and sequentiality [36].

**Branching factor:** This indicates the number of sub-problems that can be generated from dividing the main problem. Unfortunately the branching factor for our implementation varies across different data leading to difficulties in work load distribution.

**Balance:** The balance factor is concerned with whether the problem can be broken down into equal parts. Again this is not the case for SUDA2.

**Data-dependency:** This refers to the case where multiple use of the same memory location occurs. This can inhibit implementation of algorithms that fall under this category such as SUDA2.

**Sequentiality:** This issue is concerned with algorithms that require the flow of execution to follow a specific order. For example the sub-problems may need to be executed in a specified order and not independently. This is not the case in SUDA2.
6.4.3 SUDA2-Specific Parallel Issues

The parallel version of SUDA2 divides the main search into a set of sub-searches and executes each one simultaneously (assuming enough processors are provided). There are three main issues involved in the effective execution of parallel SUDA2 [36]:

**Granularity:** Granularity, in this context, is the amount of computation needed for processing a single sub-search. The minimum parallel execution time of the algorithm is the length of the longest running sub-search when granularity is fixed. This is still true even in the case of infinite available processors. Given a fixed granularity it is impossible to execute faster than that.

**Sub-search-to-Processor ratio:** The optimal case in the presence of multiple processors would be to have a high sub-search-to-processor ratio. That is, when there is sufficient number of sub-searches to maintain the processors busy while they all complete. Failure to utilize all processors into a full extend results in inefficient load balancing, meaning that some processors are wasting useful time by remaining idle and waiting for others to finish execution. Performance can be increased when work is evenly distributed among available processors.

**Spawn-Run-Return cycle:** This refers to the time needed to spawn a new sub-search, run it on a different processor and return the results. The granularity of sub-tasks directly affects this issue. A first suggestion would have been to reduce granularity at its finer form in order to execute faster. That is, by breaking down the search into sub-searches seems to produce greater number of smaller jobs to be distributed across the processors thus facilitating load balancing and maximizing the degree of concurrency. However this idea implies high communication overheads as there are more messages to be sent and received. The spawn-run-return cycle of a sub-search is significant, especially in architectures where communication and start-up latency cost is high, since when granularity is too fine there is the possibility that the start-up time and communication take longer to run than computing the sub-search locally. This issue sometimes imposes a limit on minimum granularity.
6.4.4 Implementation

Having discussed about the issues concerned with parallel SUDA2 it is important to discuss some implementation details.

As far as the data replication is concerned, a copy of each dataset is sent to each processor since each one operates in its local memory in a message-passing paradigm. In contrast to other types of applications the datasets are not massively large since we operate on microdata. Thus, creating a separate copy for each processor will not have a major effect in performance or memory availability. For this reason it is not required to consider partitioning on input data. Partitioning the dataset would have had negative effects as all candidates would need to be broadcasted around for the special row condition check.

As already mentioned, it is possible to divide the search space of the algorithm into non-overlapping sub-searches.

The way this is done is by considering a separator [36], which consists of a subset of nodes in the recursive search tree. The separator detaches this subset of nodes from the remaining tree in a way that prevents any node below the separator to have a path to the root of the tree. Furthermore, no two nodes in the separator are permitted to have an ancestor-descendant relationship. This is implemented in the easiest way which is to select as a separator, all the nodes of a specific level. This guarantees that every node of the separator is a completely independent sub-search. Selecting all the nodes at a particular level \( l \), for \( l > 0 \), is referred to as level \( l \) work generation [36].

Level 1 generation is the simplest way to perform this; however, this will produce coarse granularity parallelism which is possible to have negative effects upon load balancing.

Figure 6.2 (shown in the next page) depicts a fraction of a tree structure that Level 1 work generation is applied.
The marked nodes are those that will be disconnected from the tree and each one will be assigned on a different processor. It is easy to observe that descending down at each level the number of nodes generated can be dramatically increased. Although the tree chosen for in this case appears to be quite symmetric, it does not always reflect the reality as each node may carry unpredicted amount of computation.

A *master-worker* model will be used in which a control program named *master* is responsible for creating, initializing, and assigning jobs to other programs named *worker* programs. After the workers complete the computation of their assigned job, the master gathers and displays the results.

As soon as the program *suda2mpi* (the class that includes *main*) runs, it calls the `MPI_Init` function in order to initialize the MPI environment. It then decides whether to start a master or a worker.

Figure 6.3, shows an example of level 1 work generation based on the dataset used in Chapter 5 (Figure 5.2) where each processor in the message-passing parallel
environment is provided with the dataset $T$ and an anchor which indicates a search in a sub-table. In the case of level $l$ work generation, where $l > 1$ each processor will send the sub-search created from its anchor to a different available processor and so on. This results in each processor to operating on different levels of recursion.

**Figure 6.3:** Distribution of jobs for level 1 work generation in parallel SUDA2

**The Master Program:**

The master will first read the dataset which will be broadcasted across all participating worker processors using the $\textit{MPI\_Bcast}$ function which basically sends any data stored at the calling processor’s buffer to all other processors in the group. Along with the dataset, some other important information is also broadcasted including the level of work generation as well as the maximum number of MSUs to be searched; all of this information being supplied by the user. After that, the master checks whether there is only one processor in the group in which case an instance of search is created using the code from the serial version of SUDA2. Otherwise the parallel search is being instantiated. In the case of running the parallel search, the master program distributes the anchors across available processors using the $\textit{MPI\_Send}$ function.

Results are gathered after each worker has finished processing its assigned job, and displayed on standard output.

**The Worker Program**

When a worker is being called, it executes an $\textit{MPI\_Recv}$ indicating that it is free to be allocated work. As soon as data has been sent to it (data decomposition), the
worker starts looking for MSUs on the anchor being assigned to search. Each worker processor will apply SUDA2 into its anchor. Consider again the dataset given as an example in Chapter 5 (Figure 5.2). For instance, a processor $P_i$ working in anchor $E_2$ (Figure 6.4) will work on rows containing $E_2$. As far as recursive decomposition is concerned, each time recursion is called to operate on a sub-search (eg. table $T_{E2C1}$ in (Figure 6.4) we can send this sub-table on a different processor $P_j$ with the aim of reducing the granularity of searches.

The class responsible to handle the actions to be taken by the worker is named $MPIWorker$. Immediately after the search has finished, the worker sends a message back to the master including any MSUs found as well as notifies that it is ready to be allocated another job (if there is one waiting).

![Figure 6.4: Sub-table $T_{E2C1}$ is extracted from $T_{E2}$](image)

**Work allocation**

Work allocation plays a vital role in the whole process as it may directly affect load balancing. *Efficient work allocation may facilitate load balancing* [1, 2, 19, 20]. Since SUDA2 can generate work of unpredictable size, it is wiser to process jobs that require heavy computation first. By operating this way, all processors are guaranteed to be busy for the most part of the process. Towards the end of the process there is the case that there are not enough jobs to utilize all processors; however, this will not have a significant effect as the last jobs will also be the
smallest ones. Therefore, an unbalanced workload may exist only for a very small period of time relatively to the duration of the whole process.

Parallel SUDA2 uses the observation that usually the largest workload appears towards the middle of the process in order to allocate work efficiently [36]. Thus the algorithm starts from the middle of the tree, descends down to the requested work generation level and then begins work allocation from the middle moving toward the outside. This strategy is referred to as middle-out [36]. The reason why this “phenomenon” is observed is due to the fact that at the beginning there is not as much work to be done as we are looking at 2-MSUs (since the ITEM-LIST is sorted in ascending order of support count), and towards the end most MSUs have been discovered (and also we do not consider items that appear before the current item in the list), thus again no much work is needed.

**Adjusting computation and communication**

Our goal is to increase the average degree of concurrency via formulating finer granularity. The way to achieving this is not always straightforward. In fact by reducing the granularity the algorithm appears to slow down significantly in the presence of small number of processors. The use of Level 2 work generation shows the best possible speedup we can obtain in most cases. Level 3 work generation is even slower than Level 1 work generation in some cases. The reason why these obscure results occur is obvious. By reducing the granularity level, some tasks become so fine (small) that it is not worth to be sent for processing in a different processor. In other words some tasks are so small that their startup and communication (send/receive) time takes more than the actual computation time for a specific task. Thus a reasonable adjustment has to be achieved in order to increase efficiency. Experimental results so far, show that most of the work appears to be accumulated towards the middle of the searching process leaving the beginning and end with just a little bit of work to be done. For that reason we have decided that whenever we descend down the tree, while working on a specific work generation level, we will only take into consideration nodes that are found only in the middle of the tree. Any remaining nodes that fall outside the borders of the middle will just be
allocated to processors as level 1 work generation. This process is depicted in Figure 6.5 (shown in the next page).

Figure 6.5: An example of a work generation based on workload

As seen in the diagram, the separator now has a Π shape. The searching function will only descent and generate level $l$ work for the nodes being enclosed inside the separator. For any remaining nodes that are outside of the separator, the searching process will not descent. It will instead remain at Level 1 and generate work as appropriate. This has as an effect of creating finer granularity for work that can amplify the degree of concurrency and coarser granularity for work that can have a negative impact on concurrency (due to computation/communication imbalance).
6.5 Summary

The parallel program design was described in this chapter. Parallel designed issues, especially those related to load balancing and have significant impact upon the algorithm were also discussed. SUDA2 is a divide-and-conquer algorithm which is very effective when used in recursive decomposition parallelism. The issue of work allocation which is directly related with load balancing has been discussed extensively. A new way has been identified for more effective work allocation. The key idea is to work with finer levels of granularity towards the middle where the largest computational units are located, and work with coarser levels of granularity for very small jobs. The message passing functionality was provided by the MPI library. The MPI library provides bindings for important languages such as C and C++ and enables a collection of interconnected heterogeneous computer systems to communicate with each other.
Chapter 7

Performance Evaluation

7.1 Introduction

Several experiments have been contacted in order to evaluate the performance and scalability of the parallel system. Description about the data and the system being used, experimental results and performance analysis form the contents of this chapter.

However, performance evaluation would be incomplete if we exclude from discussion issues that do not occur in sequential programming such as sources of parallel overhead and scalability. Furthermore, this chapter will describe issues concerning parallel performance metrics. Such measurements quantify the benefit of parallelism and will be presented.

7.2 Sources of Overhead in Parallel Programs

When doubling the number of processors in a parallel environment, it is reasonable to expect the program to run twice as fast. However in reality this is rarely the case. The reasons for this are certain overheads associated with parallelism. The major
sources of overhead in parallel programs according to [19] are: *Inter-process Interaction, and Idling.*

**Inter-process Interaction:** This is usually the most significant source of parallel overhead. Any nontrivial parallel system involves some level of interaction between processors. It appears to be time consuming in order to transform the data into a form that is suitable for transmission. Encoding a message before executing a `send` function and decoding that message upon arrival may degrade the performance of the system.

**Idling:** There are mainly three issues concerned with idling: *load imbalance, synchronization, and partial parallelization.*

There are numerous situations where it is very difficult to (if not impossible) to predict the size of subtasks to be generated and assigned among processors. This result in some processors required to spend longer time in performing computations than others. The problem is that processors with smaller amounts of work will finish earlier and remain idle while some other processors keep working hard to finish the process. Such an example was presented in the previous chapter in SUDA2. Being unable to divide evenly the tasks among processors statically leads to non-uniform workload or load imbalance.

Some parallel programs require the processors to stop execution at a particular point and synchronize with each other. In case not all processors have finished execution in order to synchronize with others, the ones that are ready for synchronization will have to wait for them to finish causing them to become idle.

Finally there are cases where some parts of the initial serial algorithm cannot be parallelized. Unfortunately this can utilize a single processor to work on that serial part where other elements will work on the remaining parallel part. The processors working on the parallel part will soon finish and remain idle while a single processor may still be working on the serial part. This imposes negative implications on the whole execution time of the algorithm.

Fortunately SUDA2 does not have to worry neither for synchronization or for partial parallelization as the sub-searches are totally independent from each other.
7.3 Performance Metrics for Parallel Systems

It is very important to make use of some metrics in order to analyze the performance of a parallel system. Such metrics will be described in this section.

7.3.1 Execution Time

The serial run-time of a program is the time elapsed between the beginning and the end of its execution on a uni-processor system. A parallel run-time of a program is the time elapsed between the beginning and the end of its execution on a multi-processor environment. That is, parallel run-time is the time from the moment the first processor starts executing until the moment the last processor finishes its execution.

Serial run-time is indicated using $T_s$ and parallel run-time using $T_p$.

7.3.2 Speed-up

Speed-up, $S$, is defined as “the ratio of the serial run-time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on $p$ processors” [19, 20]. In other words speed-up refers to how much faster a parallel algorithm is able to solve a problem to its corresponding serial algorithm.

Mathematically speed-up is given by the formula

$$S = \frac{T_s}{T_p}$$

There exists the case in which more than one serial algorithm is in present. In such cases, the “best sequential algorithm” is considered to be the one that performs faster relatively to its other competitors.
In theory, speed-up can never exceed the number of processors $p$. This is due to the fact that a serial algorithm takes $T_s / p$ to solve the problem. If a parallel algorithm could reach a speed-up greater than $p$ this means that the serial algorithm could have actually performed with speed-up greater than $T_s / p$. The rationale behind the emergence of this contradiction is because speed-up by definition is computed with respect to the best serial algorithm.

Generally, increasing the number of processors does not guarantee a proportional speed-up due to parallel overheads that may be present as discussed in section 7.2. In some extreme cases a phenomenon known as Super Linear Speed-up is observed. This is the case where the parallel algorithm actually performs faster than the number of processors $p$. This is often confusing as it contradicts what has just been suggested. However, the reason why this incident occurs is usually due to hardware issues rather than algorithmic. By clustering many computers together not only the aggregate processor power increases but also the size of aggregate cache memory. This enables more data to fit into caches, resulting into faster memory access time(s) which ultimately amplifies the speed-up gained from the actual computation.

### 7.3.3 Efficiency

A speed-up that is equal to $p$ can only be obtained by an ideal parallel system with $p$ processors. In reality this is rarely the case due to the fact that processors usually cannot dedicate 100 percent of their time computing the algorithm. In section 7.2 we have seen that reasons such as idling can prevent a processor from devoting all of its time in computations.

Efficiency $E$, is “a measure of the fraction of time for which a processor usefully employed”; it is defined as “the ratio of speed-up to the number of processors” [20]. Ideally speed-up should be equal to $p$, and efficiency equal to one.

In practice (depending on the degree of effectiveness with which processors are utilized), speed-up is less than $p$, and efficiency between zero and one.

Mathematically efficiency is given by the formula

\[
E = \frac{S}{p}
\]
7.4 Scalability of Parallel Systems

In order to evaluate performance of parallel systems, there are two factors to be taken into consideration. These are computation and communication. Performance gains can be achieved either by increasing the computation performed or by decreasing communications.

There are two important observations in the majority of parallel systems [19, 24]. Firstly, the phenomenon of decrease in speed-up, while the number of processors increases has been observed. This occurs due to communication and synchronization overheads associated with the increase in processors. This also has a negative impact upon efficiency. The second observation is that higher speed-up and efficiency is achieved, considering a larger instance of the same problem using the same number of processors due to the increased amount of computation.

A scalable parallel system is one that can achieve equilibrium between those two with the aim of gaining efficiency. In other words a scalable parallel system is one that can maintain efficiency constant while simultaneously increasing the number of processors and the problem size. The scalability of a parallel system is “a measure of its capacity to increase speed-up in proportion to the number of processors” [19]. It reflects a parallel system’s ability to utilize increasing processing resources effectively in order to solve a problem of larger size [24].

In order to evaluate the scalability of a parallel system it is required to define the problem size which in most cases is a function of input size. However, when mining for MSUs the problem size is not only composed by the number of transactions and the average transaction size (which determine the input size). Other factors such as the size of MSUs and the number of MSUs also affect the problem size. These factors can directly affect the computation required as they influence the number of passes through the dataset, the number of candidates to be checked and pruned.

Furthermore factors such as these can also cause increase in communication levels. Due to substantial difficulties in determining the problem size, it is difficult to evaluate a parallel system for mining for MSUs in terms of scalability.
7.5 Datasets Used in experiments

Three datasets are used in this work: Connect-4, PUMS, and Synthetic Transactional. Based on the results obtained we measure parallel SUDA2 in terms of speedup, efficiency, and scalability. The three datasets are different in terms of size, nature, work level generation and number of sub-searches. Thus we expect them to have different run-time characteristics when applied with parallel SUDA2.

The same datasets were used to perform the experiments in [36] in order to compare the performance of the sequential and parallel SUDA2.

Connect-4 would have been sufficient to run the experiments as it presents a typical case on which the algorithm is to be applied. The Synthetic Transactional dataset is used in order to experience performance results in an environment of a different nature. Finally the PUMS dataset was chosen to show the impact on performance on a case that data does not follow the trend expected by the algorithm. In other words PUMS data does not accumulate work towards the middle of the process as other datasets do.

Detail concerning the nature of the three datasets is provided in the next sections:

7.5.1 Connect-4 Dataset

The Connect-4 dataset shows evidence of SUDA2’s performance on a dataset that is being used as a standard for experiments in the UCI Machine Learning Repository [37]. It can very easily be applied in this case as it shares the same characteristics as with the case of microdata. Furthermore it is very suitable as it presents none of the ordinary confidentiality restrictions that apply. The dataset reflect the game named Connect-4 and contains all 8-ply positions where the next step is not forced, and none of the players has won yet. The number of “transactions” is about 70,000 and there are 43 attributes per transaction.
7.5.2 PUMS Dataset

This dataset represents information gathered from the US 2000 Census [38]. It covers topics including lifestyles, incomes, jobs, etc. Small samples, named Public Use Microdata Samples (PUMS) were extracted from the initial information to form the dataset used in this experiment. In fact it is the type of sample expected for the statistical disclosure risk assessment application of SUDA2. It is comprised by 67,625 rows and 100 columns. Our experiments were restricted to \( maxk = 4 \) in order to allow reasonable execution times, as the number of anchors generated is excessively large compared to the Connect-4 dataset.

7.5.3 Synthetic Transactional Dataset

This dataset was generated\(^2\) in order to provide an idea of how parallel SUDA2 performs in an environment of different nature. Essentially this data imitates transactions of a retail form by generating a set of transactions along with a set of items bought in each transaction [8]. This is the type of data used in market basket analysis. A transformation of representing a purchase of an item by a 1, and no purchase of an item with a 0 was applied in order to reflect the input format for SUDA2. The dataset contains 100,000 transactions each restricted to 100 items.

7.6 Experimental Results

The MPI Experiments were performed using the High-Performance Computer at MSU (Minnesota State University) which is an SMP Cluster that has the following specifications:

- There is one master node and 34 worker nodes
- Each node has 8 GB of RAM

\(^2\) The generation software for synthetic datasets is available at [http://www.almaden.ibm.com](http://www.almaden.ibm.com)
• Each node has two dual-core AMD Opteron 270 2.0 GHz
• All nodes run the Linux operating system

SUDA2 was compiled using GNU g++ v3.4.4 and linked with OpenMPI v1.1.2 (an implementation of the MPI-2 standard).

The \textit{maxk}, which is the maximum size of a MSU, is supplied to all experiments, as a parameter, in order to allow reasonable execution times for all number of processors (especially small numbers).

The next pages will present charts showing the obtained results in terms of execution time, speedup, and efficiency. Furthermore analysis will follow concerning the different experiments along with any results and observations.

In some cases no experiments were run on relatively lower levels of work generation due to the high number of sub-searches being produced resulting in incredibly long execution time.

\section*{7.7 Performance Analysis}

The optimization method used, as explained in Chapter 6, appears to achieve significantly higher performance gains than before.

However this is true only for that specific level where the optimization is applied. That is, the optimized algorithm does have a greater performance compared to the fastest parallel algorithm. It optimizes the current level on which it is applied to work. The level in which optimization is applied is indicated using the \textit{Opt} prefix.

The words \textit{task} and \textit{search} have the same meaning in this context, and thus, are going to be used interchangeably for the remaining of this Chapter (i.e. sub-search or sub-task).
Figure 7.1: Execution time for Connect-4 dataset with $maxk = 5$

Figure 7.2: Speedup for Connect-4 dataset with $maxk = 5$

Figure 7.3: Efficiency for Connect-4 dataset with $maxk = 5$
Chapter 7

Performance Evaluation

7.7.1 Analysis using Connect-4 dataset with $maxk = 5$

In this experiment the Connect-4 dataset was used supplying $maxk = 5$ in order to realize performance gains on a reasonably small number of MSUs. A total of 145,414 MSUs have been identified in this experiment. Optimization was applied for both Level 2 and Level 3 work generation.

Level 1 work generation has obtained the best performance results. The small size and number of MSUs to be identified here, result in smaller amounts of work which make work generation in lower granularities unnecessary. That is, the tasks are so small, that communication will be more costly if they are broken down to sub-tasks. When optimization is applied at Level 2 and Level 3 work generation (for their normal corresponding levels) lower execution time is observed (see Figure 7.1), thus higher speedup (see Figure 7.2).

**Speedup Analysis**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Speedup</th>
<th>Level 2 Speedup</th>
<th>Opt_Level 2 Speedup</th>
<th>Level 3 Speedup</th>
<th>Opt_Level 3 Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.48</td>
<td>0.85</td>
<td>1.09</td>
<td>0.06</td>
<td>0.55</td>
</tr>
<tr>
<td>8</td>
<td>5.27</td>
<td>1.79</td>
<td>2.30</td>
<td>0.12</td>
<td>1.18</td>
</tr>
<tr>
<td>12</td>
<td>7.97</td>
<td>2.66</td>
<td>3.47</td>
<td>0.18</td>
<td>1.79</td>
</tr>
<tr>
<td>16</td>
<td>10.17</td>
<td>3.55</td>
<td>4.54</td>
<td>0.24</td>
<td>2.38</td>
</tr>
<tr>
<td>20</td>
<td>11.80</td>
<td>4.34</td>
<td>5.46</td>
<td>0.30</td>
<td>2.89</td>
</tr>
<tr>
<td>24</td>
<td>13.41</td>
<td>5.18</td>
<td>6.41</td>
<td>0.37</td>
<td>3.43</td>
</tr>
<tr>
<td>36</td>
<td>18.44</td>
<td>7.38</td>
<td>8.43</td>
<td>0.55</td>
<td>5.00</td>
</tr>
<tr>
<td>48</td>
<td>21.07</td>
<td>8.94</td>
<td>10.54</td>
<td>0.72</td>
<td>6.15</td>
</tr>
<tr>
<td>60</td>
<td>21.07</td>
<td>10.93</td>
<td>11.80</td>
<td>0.89</td>
<td>7.20</td>
</tr>
<tr>
<td>72</td>
<td>24.58</td>
<td>12.29</td>
<td>12.83</td>
<td>1.05</td>
<td>8.43</td>
</tr>
<tr>
<td>84</td>
<td>24.58</td>
<td>12.83</td>
<td>13.41</td>
<td>1.20</td>
<td>8.94</td>
</tr>
<tr>
<td>96</td>
<td>26.82</td>
<td>14.05</td>
<td>15.53</td>
<td>1.36</td>
<td>10.17</td>
</tr>
<tr>
<td>108</td>
<td>22.69</td>
<td>15.53</td>
<td>17.35</td>
<td>1.50</td>
<td>10.54</td>
</tr>
<tr>
<td>120</td>
<td>21.07</td>
<td>16.39</td>
<td>16.39</td>
<td>1.63</td>
<td>10.93</td>
</tr>
<tr>
<td>132</td>
<td>24.58</td>
<td>18.44</td>
<td>18.44</td>
<td>1.76</td>
<td>11.35</td>
</tr>
</tbody>
</table>

**Figure 7.4:** Speedup results for Connect-4 with $maxk = 5$

Opt_Level 2 shows greater speedups compared to Level 2 for up to 108 processors as illustrated in Figure 7.2. The exact speedup results obtained are shown in Figure 7.4.
For instance using 8 processors, Opt_Level 2 obtains a speedup of 2.3 whereas Level 2 obtains a speedup of 1.79. For 108 processors Opt_Level 2 has speedup 17.35 compared to 15.53 of Level 2. Between 120 and 132 processors both Opt_Level 2 and Level 2 have exactly the same speedup. The average speedup for Opt_Level 2 and Level 2 is 8.7 and 7.71 respectively, between 4 and 108 processors. Opt_Level 3 appears to be several orders of magnitude faster than Level 3. For example using 16 processors Opt_Level 3 has speedup 2.38 whereas Level 3 has only 0.24. For 132 processors Opt_Level 3 shows a speedup of 11.35 whereas Level 3 shows a speedup of 1.76. The average speedup for Opt_Level 3 from 4 to 132 processors is 6.06 which is significantly greater compared to the average speedup of Level 3 which is only 0.79.

Although Level 1 has greater speedup compared to Opt_Level 2 and Opt_Level 3, a decline is observed (for Level 1) for more than 96 processors.

**Efficiency Analysis**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Efficiency</th>
<th>Level 2 Efficiency</th>
<th>Opt_Level 2 Efficiency</th>
<th>Level 3 Efficiency</th>
<th>Opt_Level 3 Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>61.97%</td>
<td>21.19%</td>
<td>27.31%</td>
<td>1.39%</td>
<td>13.86%</td>
</tr>
<tr>
<td>8</td>
<td>65.85%</td>
<td>22.35%</td>
<td>28.81%</td>
<td>1.48%</td>
<td>14.69%</td>
</tr>
<tr>
<td>12</td>
<td>66.44%</td>
<td>22.15%</td>
<td>28.92%</td>
<td>1.50%</td>
<td>14.90%</td>
</tr>
<tr>
<td>16</td>
<td>63.58%</td>
<td>22.21%</td>
<td>28.37%</td>
<td>1.52%</td>
<td>14.87%</td>
</tr>
<tr>
<td>20</td>
<td>59.00%</td>
<td>21.69%</td>
<td>27.31%</td>
<td>1.52%</td>
<td>14.46%</td>
</tr>
<tr>
<td>24</td>
<td>55.87%</td>
<td>21.56%</td>
<td>26.72%</td>
<td>1.53%</td>
<td>14.29%</td>
</tr>
<tr>
<td>36</td>
<td>51.22%</td>
<td>20.49%</td>
<td>23.41%</td>
<td>1.51%</td>
<td>13.89%</td>
</tr>
<tr>
<td>48</td>
<td>43.90%</td>
<td>18.62%</td>
<td>21.95%</td>
<td>1.50%</td>
<td>12.80%</td>
</tr>
<tr>
<td>60</td>
<td>35.12%</td>
<td>18.21%</td>
<td>19.67%</td>
<td>1.48%</td>
<td>11.99%</td>
</tr>
<tr>
<td>72</td>
<td>34.14%</td>
<td>17.07%</td>
<td>17.81%</td>
<td>1.46%</td>
<td>11.71%</td>
</tr>
<tr>
<td>84</td>
<td>29.27%</td>
<td>15.27%</td>
<td>15.96%</td>
<td>1.43%</td>
<td>10.64%</td>
</tr>
<tr>
<td>96</td>
<td>27.94%</td>
<td>14.63%</td>
<td>16.17%</td>
<td>1.42%</td>
<td>10.60%</td>
</tr>
<tr>
<td>108</td>
<td>21.01%</td>
<td>14.38%</td>
<td>16.07%</td>
<td>1.39%</td>
<td>9.76%</td>
</tr>
<tr>
<td>120</td>
<td>17.56%</td>
<td>13.66%</td>
<td>13.66%</td>
<td>1.36%</td>
<td>9.10%</td>
</tr>
<tr>
<td>132</td>
<td>18.62%</td>
<td>13.97%</td>
<td>13.97%</td>
<td>1.33%</td>
<td>8.60%</td>
</tr>
</tbody>
</table>

**Figure 7.5:** Efficiency results for Connect-4 with $maxk = 5$

The system also seems to be utilized more efficiently for Opt_Level 2 and Opt_Level 3 as illustrated in Figure 7.3 and Figure 7.5.

Level 2 spends on average 18.5% of execution in useful computation (i.e. searching) with a total of 81.5% spent as parallel overhead. A 3.24% increase is observed as
Opt_Level 2 spends on average 21.74% of execution time in useful computation and 78.26% in parallel overhead. Parallel overhead is caused from extra computation for data conversion, serial components of the algorithm that cannot be parallelized, and mainly inter-process communication [24].

Significant increase in efficiency levels is observed between Opt_Level 3 and Level 3. Level 3 spends on average only 1.45% of execution time in useful computation and a total of 98.55% in parallel overhead. Opt_Level 3 spends 12.41% of execution in useful computation and 87.59% in parallel overhead.

Although Level 1 still seems to outweigh all other levels, it fluctuates to a large extent. Where Opt_Level 2 and Opt_Level 3 show scalability up to 100+ processors, Level 1 seems to scale up to 36 processors and then dramatically falls. This observation, as we will see in the next experiment, will have a negative impact for Level 1 work generation when searching for larger (in terms of size and number) MSUs.
Figure 7.6: Execution time for Connect-4 dataset with $maxk = 12$

Figure 7.7: Speedup for Connect-4 dataset with $maxk = 12$

Figure 7.8: Efficiency for Connect-4 dataset with $maxk = 12$
7.7.2 Analysis using Connect-4 dataset with $maxk = 12$

The same dataset as the one used in the previous experiment is now processed using $maxk = 12$. This results in identification of 3,239,319 MSUs. It is noticeable that an increase the size of $maxk$ allows for a larger number of MSUs to be discovered. In the previous experiment with $maxk = 5$, 145,414 MSUs were found whereas increasing $maxk$ to 12 caused the discovery of 3,093,905 additional MSUs. By increasing the size of $maxk$, we allow for larger sets of attributes to be checked for uniqueness. That is, the larger the sets to be considered for checking, the more unique combinations there exist. Consequently this has a negative impact upon execution time meaning that the more MSUs to be discovered, the longer it takes for the process to be executed. For this reason $maxk$ was provided with values that allow reasonable execution times, especially for small number of processors.

The optimization applied only at Level 3 in this case. Large values of $maxk$ imply more MSUs to be discovered. More MSUs imply that more processing is needed and therefore optimizing at high levels of work generation such as Level 1 and Level 2 will not provide any gains. This is due to the fact that the tasks are too large and need to be divided in smaller pieces in order take advantage of the parallel architecture. Dividing the tasks into smaller pieces requires descending to lower levels of the recursion tree and producing work generation at these levels. For this reason optimization works better in lower levels of work generation in this case. Furthermore, optimizing Level 4 is worthless as already Level 3 is poor compared to its corresponding Level 2. Thus Level 3 was chosen to be optimized.
Speedup Analysis

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Speedup</th>
<th>Level 2 Speedup</th>
<th>Level 3 Speedup</th>
<th>Opt_ Level 3 Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.92</td>
<td>2.82</td>
<td>1.79</td>
<td>2.75</td>
</tr>
<tr>
<td>8</td>
<td>6.39</td>
<td>6.46</td>
<td>3.92</td>
<td>6.34</td>
</tr>
<tr>
<td>12</td>
<td>9.60</td>
<td>10.13</td>
<td>6.18</td>
<td>9.91</td>
</tr>
<tr>
<td>16</td>
<td>12.93</td>
<td>13.77</td>
<td>8.22</td>
<td>13.45</td>
</tr>
<tr>
<td>20</td>
<td>16.08</td>
<td>17.19</td>
<td>10.56</td>
<td>16.87</td>
</tr>
<tr>
<td>24</td>
<td>18.25</td>
<td>20.32</td>
<td>12.78</td>
<td>20.08</td>
</tr>
<tr>
<td>36</td>
<td>22.27</td>
<td>29.10</td>
<td>18.84</td>
<td>29.06</td>
</tr>
<tr>
<td>48</td>
<td>22.46</td>
<td>37.66</td>
<td>25.72</td>
<td>36.70</td>
</tr>
<tr>
<td>60</td>
<td>22.33</td>
<td>42.84</td>
<td>32.10</td>
<td>43.73</td>
</tr>
<tr>
<td>72</td>
<td>22.55</td>
<td>51.79</td>
<td>38.48</td>
<td>50.65</td>
</tr>
<tr>
<td>84</td>
<td>22.35</td>
<td>58.35</td>
<td>44.84</td>
<td>57.33</td>
</tr>
<tr>
<td>96</td>
<td>22.62</td>
<td>62.29</td>
<td>51.10</td>
<td>62.12</td>
</tr>
<tr>
<td>108</td>
<td>22.46</td>
<td>68.19</td>
<td>57.19</td>
<td>67.59</td>
</tr>
<tr>
<td>120</td>
<td>22.40</td>
<td>73.63</td>
<td>63.14</td>
<td>72.02</td>
</tr>
<tr>
<td>132</td>
<td>22.46</td>
<td>77.08</td>
<td>69.21</td>
<td>75.07</td>
</tr>
</tbody>
</table>

Figure 7.9: Speedup results for Connect-4 with maxk = 12

Level 1 shows higher speedup compared to other levels only when using 4 processors (see Figure 7.7 and Figure 7.9). It shows a linear increase in speedup for up to 36 processors where it flattens. As already mentioned earlier in this dissertation, this experiment identifies a relatively large number of MSUs and thus, larger tasks to be processed. For a small number of processors such as 4, producing finer granularity shows no gains as there are not enough processors (compared to the number of sub-tasks) to be utilized in an effective manner. While increasing the number of processors, the load imbalance created due to the fact that large pieces of tasks are not manipulated effectively, prevents Level 1 from gaining additional speedup.

Level 2 shows great scalability for 100+ processors. Although Level 3 is also scalable to a 100+ processors, it appears to be poorer compared to its corresponding Level 2. This is due to the reason that sub-tasks were already at a reasonably small size at Level 2 and therefore any further decomposition generated at Level 3 resulted in additional overhead due to inter-processes communication. That is, a number of tasks at Level 3 are so small that the spawn-run-return cycle (see section 6.4.3) is large enough to cause a negative impact on the performance of the algorithm.
Level 3 has an average speedup of 29.6 and Level 2 an average speedup of 38.11. Opt_level 3 prevents further decomposition of tasks that are already small enough and therefore demonstrates much higher speedup compared to Level 3. We can observe a significant increase in speedup from 29.6 (Level 3) to 37.58 (Opt_level 3). Figure 7.9 illustrates that for all number of processors, the speedup achieved from optimization was significantly higher than before. Results obtained from Opt_Level 3 are even comparable (and better in some cases) with Level 2. For instance using 60 processors Opt_Level 3 (speedup 43.73) outperforms Level 2 (speedup 42.84).

**Efficiency Analysis**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Efficiency</th>
<th>Level 2 Efficiency</th>
<th>Level 3 Efficiency</th>
<th>Opt_Level 3 Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>73.01%</td>
<td>70.42%</td>
<td>44.69%</td>
<td>68.81%</td>
</tr>
<tr>
<td>8</td>
<td>79.91%</td>
<td>80.74%</td>
<td>48.94%</td>
<td>79.30%</td>
</tr>
<tr>
<td>12</td>
<td>80.02%</td>
<td>84.38%</td>
<td>51.48%</td>
<td>82.61%</td>
</tr>
<tr>
<td>16</td>
<td>80.83%</td>
<td>86.05%</td>
<td>51.39%</td>
<td>84.04%</td>
</tr>
<tr>
<td>20</td>
<td>80.42%</td>
<td>85.93%</td>
<td>52.81%</td>
<td>84.36%</td>
</tr>
<tr>
<td>24</td>
<td>76.03%</td>
<td>84.68%</td>
<td>53.26%</td>
<td>83.65%</td>
</tr>
<tr>
<td>36</td>
<td>61.85%</td>
<td>80.83%</td>
<td>52.35%</td>
<td>80.73%</td>
</tr>
<tr>
<td>48</td>
<td>46.80%</td>
<td>78.46%</td>
<td>53.59%</td>
<td>76.46%</td>
</tr>
<tr>
<td>60</td>
<td>37.22%</td>
<td>71.40%</td>
<td>53.50%</td>
<td>72.89%</td>
</tr>
<tr>
<td>72</td>
<td>31.32%</td>
<td>71.93%</td>
<td>53.44%</td>
<td>70.35%</td>
</tr>
<tr>
<td>84</td>
<td>26.61%</td>
<td>69.46%</td>
<td>53.38%</td>
<td>68.25%</td>
</tr>
<tr>
<td>96</td>
<td>23.56%</td>
<td>64.88%</td>
<td>53.23%</td>
<td>64.71%</td>
</tr>
<tr>
<td>108</td>
<td>20.80%</td>
<td>63.14%</td>
<td>52.95%</td>
<td>62.58%</td>
</tr>
<tr>
<td>120</td>
<td>18.66%</td>
<td>61.36%</td>
<td>52.62%</td>
<td>60.02%</td>
</tr>
<tr>
<td>132</td>
<td>17.02%</td>
<td>58.39%</td>
<td>52.43%</td>
<td>56.87%</td>
</tr>
</tbody>
</table>

**Figure 7.10:** Efficiency results for Connect-4 with \( \text{maxk} = 12 \)

Level 1 appears to utilize the system efficiently only for up to 12 processors with approximately 80% of the execution spent in useful computation. It then flattens for a while between 12 and 20 processors, and then starts a rapid decrease (from 80.42 at 20 processors to 17.02 at 132 processors). This negative effect is caused by the small sub-search-to-processor ratio (see section 6.4.3) as shown in Figure 7.8. That is, the tasks are too large at Level 1 work generation causing the system to have insufficient number of sub-tasks to maintain the processors busy while they all complete.

At Level 2 efficiently levels reach 74.14% (25.86% parallel overhead) on average, with the highest level being 86.05% (13.95% parallel overhead) using 16 processors.
A small decline as shown in Figure 7.8 is observed after 16 processors which cause efficiency levels to drop to 58.39%. (41.61% parallel overhead).

While Level 3 maintains very stable efficiency levels, it does not utilize the parallel system as desired. It shows a small increase from 44.69% to 53.26% between 4 and 24 processors and then flattens.

By applying the optimization technique, Opt_Level 3 managed to improve efficiency levels for all number of processors as illustrated in Figure 7.8 and Figure 7.10. Opt_Level 3 spends on average 73.04%, of total execution, performing useful computation (the remaining 26.96% denotes parallel overhead) while Level 3 spends only 52% (48% parallel overhead) on average in useful computation. Between 4 and 72 processors Opt_Level 3 is approximately 20% more efficient than Level 3. Beyond 72 processors is approximately 10% more efficient.

Furthermore, Opt_Level 3 (73.04% on average) has comparable efficiency levels with Level 2 (74.14% on average).

As a matter of fact, in the case of 60 processors Opt_Level 3 spends 72.89% of the total execution time in useful computation compared to 71.4% that Level 2 does.

In conclusion, the optimization technique realizes better performance gains when it is applied in datasets with larger values for $maxk$. This is true due to the fact that there are very large tasks enabling work to be generated in lower levels and thus maintaining a high sub-search-to-processor ratio in order to keep all processors busy. Furthermore, the optimization technique prevents the decomposition of already small tasks maintaining a lower cost for the spawn-run-return cycle.
Figure 7.11: Execution time for PUMS dataset with $maxk = 4$

Figure 7.12: Speedup for PUMS dataset with $maxk = 4$

Figure 7.13: Efficiency for PUMS dataset with $maxk = 4$
7.7.3 Analysis using PUMS dataset with maxk = 4

This dataset does not show great achievements in terms of performance; however it reveals some important issues to be considered in further research. Even with maxk = 4 this dataset still produces a massive number of MSUs (485,570,215).

Due to the large number of sub-tasks (i.e. long execution time) generated at Level 2, optimization was applied only at that level. For instance, the algorithm was running for approximately 38 hours to discover all the MSUs using 4 processors at Level 2 (see Figure 7.11). Thus no further experiments were run for lower levels of work generation.

Level 1 work generation appears to obtain the fastest execution time, thus higher speedups. The next paragraphs will describe the reasons why this is observed as well as the performance gains for Level 1, Level 2, and Opt_Level 2.

Speedup Analysis

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Speedup</th>
<th>Level 2 Speedup</th>
<th>Opt_Level 2 Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.78</td>
<td>0.03</td>
<td>0.05</td>
</tr>
<tr>
<td>8</td>
<td>6.26</td>
<td>0.07</td>
<td>0.11</td>
</tr>
<tr>
<td>12</td>
<td>9.74</td>
<td>0.11</td>
<td>0.17</td>
</tr>
<tr>
<td>16</td>
<td>13.11</td>
<td>0.15</td>
<td>0.23</td>
</tr>
<tr>
<td>20</td>
<td>16.52</td>
<td>0.19</td>
<td>0.28</td>
</tr>
<tr>
<td>24</td>
<td>19.78</td>
<td>0.22</td>
<td>0.34</td>
</tr>
<tr>
<td>36</td>
<td>29.48</td>
<td>0.33</td>
<td>0.51</td>
</tr>
<tr>
<td>48</td>
<td>38.88</td>
<td>0.44</td>
<td>0.67</td>
</tr>
<tr>
<td>60</td>
<td>47.98</td>
<td>0.54</td>
<td>0.83</td>
</tr>
<tr>
<td>72</td>
<td>56.38</td>
<td>0.64</td>
<td>0.99</td>
</tr>
<tr>
<td>84</td>
<td>64.43</td>
<td>0.74</td>
<td>1.14</td>
</tr>
<tr>
<td>96</td>
<td>73.93</td>
<td>0.83</td>
<td>1.29</td>
</tr>
<tr>
<td>108</td>
<td>82.00</td>
<td>0.93</td>
<td>1.45</td>
</tr>
<tr>
<td>120</td>
<td>88.43</td>
<td>1.01</td>
<td>1.58</td>
</tr>
<tr>
<td>132</td>
<td>93.96</td>
<td>1.11</td>
<td>1.73</td>
</tr>
</tbody>
</table>

Figure 7.14: Speedup results for PUMS with maxk = 4

Figure 7.12 and Figure 7.14 show the speedup achievements for all levels. Overall, Level 1 shows great scalability. Speedup for that Level appears to increase in a linear fashion, every time more processors are added. Figure 7.11 illustrates that for 4 processors the execution time was 1625 seconds (speedup 2.78), 116 seconds...
(speedup 38.88) for 48 processors, and 48 seconds (speedup 93.96) for 132 processors.

Producing work generation at Level 2 appears to dramatically degrade the performance. Execution time for 4 processors was 134417 seconds (speedup 0.05), 10340 seconds (speedup 0.44) for 48 processors, and 4078 (speedup 1.11) for 132 processors. Although Level 2 shows great scalability, it is poor in terms of performance compared to Level 1.

Opt_LEVEL 2 demonstrates considerable improvements for Level 2. The average speedup for Level 2 was 0.49 and Opt_LEVEL 2 boosted it to 0.76. For instance, using 4 processors speedup shifted from 0.03 (Level 2) to 0.05 (Opt_LEVEL 2), using 48 processors speedup shifted from 0.44 (Level 2) to 0.67 (Opt_LEVEL 2), and using 132 processors speedup shifted from 1.11 (Level 2) to 1.73 (Opt_LEVEL 2). While Opt_LEVEL 2 shows great scalability as well, it is still poorer in terms of speedup gains compared to Level 1.

Efficiency Analysis

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Efficiency</th>
<th>Level 2 Efficiency</th>
<th>Opt_LEVEL 2 Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>69.38%</td>
<td>0.84%</td>
<td>1.27%</td>
</tr>
<tr>
<td>8</td>
<td>78.19%</td>
<td>0.88%</td>
<td>1.38%</td>
</tr>
<tr>
<td>12</td>
<td>81.17%</td>
<td>0.91%</td>
<td>1.40%</td>
</tr>
<tr>
<td>16</td>
<td>81.94%</td>
<td>0.92%</td>
<td>1.41%</td>
</tr>
<tr>
<td>20</td>
<td>82.60%</td>
<td>0.93%</td>
<td>1.42%</td>
</tr>
<tr>
<td>24</td>
<td>82.42%</td>
<td>0.92%</td>
<td>1.41%</td>
</tr>
<tr>
<td>36</td>
<td>81.88%</td>
<td>0.92%</td>
<td>1.41%</td>
</tr>
<tr>
<td>48</td>
<td>81.00%</td>
<td>0.91%</td>
<td>1.40%</td>
</tr>
<tr>
<td>60</td>
<td>79.96%</td>
<td>0.90%</td>
<td>1.39%</td>
</tr>
<tr>
<td>72</td>
<td>78.30%</td>
<td>0.89%</td>
<td>1.37%</td>
</tr>
<tr>
<td>84</td>
<td>76.70%</td>
<td>0.88%</td>
<td>1.36%</td>
</tr>
<tr>
<td>96</td>
<td>77.02%</td>
<td>0.87%</td>
<td>1.34%</td>
</tr>
<tr>
<td>108</td>
<td>75.93%</td>
<td>0.86%</td>
<td>1.34%</td>
</tr>
<tr>
<td>120</td>
<td>73.69%</td>
<td>0.85%</td>
<td>1.32%</td>
</tr>
<tr>
<td>132</td>
<td>71.18%</td>
<td>0.84%</td>
<td>1.31%</td>
</tr>
</tbody>
</table>

Figure 7.15: Efficiency results for PUMS with $\text{maxk} = 4$

Figure 7.13 and Figure 7.15 indicate that Level 1 achieves respectable efficiency levels with an average of 78.09% of total execution spent in useful computation and the remaining 21.91% spent in parallel overhead. It demonstrates an increase from
69.38% (using 4 processors) to 81% for up to 48 processors and then starts a very small decline to reach 71.18% in 132 processors.

Level 2, as well as Opt_Level 2 demonstrates poor efficiency accomplishments. Opt_Level 2 outperforms Level 2 with an average efficiency of 1.37% and 0.89% respectively. Although both levels show a linear efficiency they are completely flat demonstrating no potential for an increase at any time.

This experiment appears to contradict with our previous discussion arguing that when searching for large numbers of MSUs, a reduction in the granularity level will facilitate the process to execute faster. The main reason for this, is that the specific dataset is one of the few that does not follow the pattern being observed. That is, it appears that the greatest part of work in the process lies somewhere between the middle and right end, as indicated by [36]. Chapter 6 explained that the algorithm assumes that work is accumulated towards the middle of the process in order to select the biggest tasks first and send them for processing.

Since using the PUMS dataset, the work generated does not comply to this characteristic, the middle-out heuristic that SUDA2 employs has much less impact here. It will be unrealistic to expect the optimization method to have significant improvement since Level 2 shows such a poor performance. When work generation was applied at Level 2, the algorithm itself started distributing work to processors from the middle of the process. However, this time the middle of the process contains small sub-tasks instead of the large ones expected. Therefore small sub-tasks are assigned to processors first. Consequently a small number of large sub-tasks will remain last for processing causing load imbalance due to a low sub-search-to-processor ratio which fails to keep all the processors busy at all the time.

Furthermore since the optimization targets the middle of the tree, a substantial amount of work is left out of the partitioning. The optimization is applied at a particular level in order to further break-down tasks that are too large to be processed. In this case the middle of the process at Level 2 includes small tasks instead. Therefore those small tasks have been partitioned into even smaller sub-tasks (at Level 2) while larger tasks were left un-partitioned at Level 1. Hence the
benefits said to be gained from applying the optimization method to the algorithm were decreased due to unnecessary communication caused from the high cost of spawn-run-return cycle (resulted from those small tasks).
Figure 7.16: Execution time for Synthetic transactional dataset with $maxk = 5$

Figure 7.17: Speedup for Synthetic transactional dataset with $maxk = 5$

Figure 7.18: Efficiency for Synthetic transactional dataset with $maxk = 5$
7.7.4 Analysis using Synthetic transactional dataset with $maxk = 5$

Applying parallel SUDA2 to the Synthetic transactional dataset with $maxk = 5$, has detected 14,090,690 MSUs.

Level 2 demonstrates the best performance gains compared to all other levels. Optimization was applied at Level 3 showing significant improvement, in terms of performance gains, at that level.

**Speedup Analysis**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Speedup</th>
<th>Level 2 Speedup</th>
<th>Level 3 Speedup</th>
<th>Opt_Level 3 Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.56</td>
<td>2.03</td>
<td>0.23</td>
<td>0.99</td>
</tr>
<tr>
<td>8</td>
<td>5.64</td>
<td>4.48</td>
<td>0.27</td>
<td>2.17</td>
</tr>
<tr>
<td>12</td>
<td>8.80</td>
<td>6.92</td>
<td>0.41</td>
<td>3.33</td>
</tr>
<tr>
<td>16</td>
<td>11.88</td>
<td>9.29</td>
<td>0.55</td>
<td>4.51</td>
</tr>
<tr>
<td>20</td>
<td>15.03</td>
<td>11.75</td>
<td>0.69</td>
<td>5.66</td>
</tr>
<tr>
<td>24</td>
<td>18.04</td>
<td>14.09</td>
<td>0.83</td>
<td>6.82</td>
</tr>
<tr>
<td>36</td>
<td>27.07</td>
<td>21.22</td>
<td>1.24</td>
<td>10.26</td>
</tr>
<tr>
<td>48</td>
<td>35.98</td>
<td>28.23</td>
<td>1.67</td>
<td>13.57</td>
</tr>
<tr>
<td>60</td>
<td>40.31</td>
<td>35.12</td>
<td>2.08</td>
<td>16.77</td>
</tr>
<tr>
<td>72</td>
<td>43.77</td>
<td>42.14</td>
<td>2.49</td>
<td>19.82</td>
</tr>
<tr>
<td>84</td>
<td>44.63</td>
<td>48.55</td>
<td>2.90</td>
<td>22.86</td>
</tr>
<tr>
<td>96</td>
<td>47.20</td>
<td>55.39</td>
<td>3.31</td>
<td>25.42</td>
</tr>
<tr>
<td>108</td>
<td>48.78</td>
<td>62.31</td>
<td>3.69</td>
<td>28.87</td>
</tr>
<tr>
<td>120</td>
<td>43.30</td>
<td>68.59</td>
<td>4.08</td>
<td>31.49</td>
</tr>
<tr>
<td>132</td>
<td>50.10</td>
<td>74.87</td>
<td>4.47</td>
<td>34.07</td>
</tr>
</tbody>
</table>

**Figure 7.19**: Speedup results for Synthetic transactional with $maxk = 5$

As illustrated in Figure 7.17 and Figure 7.19, Level 1 allows great scalability and shows good speedup up to around 60 processors (2.56 for 4 processors, 40.31 for 60 processors). Beyond 60 processors speedup achieved increases with slower rates (40.31 for 60 processors, 50.10 for 132 processors) with a noticeable drop on 120 processors to 43.3. Even though the tasks are large in size, there are plenty of them (199 in total) to keep the processors busy for up to 60 processors. Beyond 60 processors we conclude that the more processors are added, the less speedup gains are achieved. This is happening due to the low sub-search-to-processor ratio. In other words the process fails to produce enough tasks to fully utilize all processors.
Level 2 obtains excellent scalability for more than 100+ processors. The more processors are added, the greater speedup is observed. The rate that the speedup increases is of a linear fashion. It is interesting that for up to 72 processors the speedup achieved is less than the one achieved at Level 1 in every case. For instance employing 4 processors Level 2 obtains a speedup of 2.03 whereas Level 1 obtains a speedup of 2.56. Using 24 processors Level 2 obtains a speedup of 14.09 whereas Level 1 obtains a speedup of 18.04. Beyond 72 processors Level 2 appears to outperform Level 1. In fact the more processors are added, the greatest the gap becomes between speedup achievements of the two levels. For example a small difference is observed for 84 processors were Level 1 has speedup 44.63 and Level 2 has speedup 48.55. Using 108 processors Level 1 has speedup 48.78 and Level 2 has speedup 62.31. Finally for 132 processors the gap increases with Level 1 obtaining speedup of only 50.10 and Level 2 speedup of 74.87. This amplifies the argument we had before for Level 1. For up to around 60-72 processors Level 1 has a relatively high sub-search-to-processor ratio. Level 2 at that point does not realize great performance gains due to the fact that processors are very small in number compared to the number of sub-tasks generated at that level and therefore excess communication creates an overhead. Beyond 60-72 processors the sub-search-to-processor ratio for Level 1 drops, causing load imbalance. Level 2 at that point starts to realize the gains from decomposing the tasks to even smaller sub-tasks creating a more balanced distribution of work.

Level 3 demonstrates very poor speedup gains, especially for a small number of processors. Level 3 decomposes the subtasks of Level 2 to even smaller sub-tasks. The fact that sub-tasks at Level 2 (for a small number of processors) were small enough to cause communication overheads, explains Level 3’s poor performance. Level 3 shows to scale in proportion to the number of processors; however, the sub-tasks are way too small and too many, even for 132 processors. Thus, there is a high cost of the time spent in spawn-run-return cycle. The increment observed in speedup (although is linear) is very small (staring at 0.23 for 4 processors and raising up to 4.47 for 132 processors).
Opt_Level 3 shows significant improvements in terms of speedup, especially as more processors are added. While Level 3 has an average speedup of 1.93, Opt_Level 3 boosts speedup to an average of 15.11. For up to 24 processors speedup appears to increase approximately 1 unit every time more processors are added. Beyond 24 processors speedup increases approximately 3 units at a time. Using 24 processors Level 3 achieves a speedup of 0.83 whereas Opt_Level 3 achieves a speedup of 6.82. Using 132 processors Level 3 achieves a speedup of 4.47 whereas Opt_Level 3 achieves the considerably better speedup of 34.07.

**Efficiency Analysis**

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 1 Efficiency</th>
<th>Level 2 Efficiency</th>
<th>Level 3 Efficiency</th>
<th>Opt_Level 3 Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>63.94%</td>
<td>50.83%</td>
<td>5.68%</td>
<td>24.69%</td>
</tr>
<tr>
<td>8</td>
<td>70.54%</td>
<td>55.98%</td>
<td>3.32%</td>
<td>27.08%</td>
</tr>
<tr>
<td>12</td>
<td>73.32%</td>
<td>57.70%</td>
<td>3.40%</td>
<td>27.72%</td>
</tr>
<tr>
<td>16</td>
<td>74.27%</td>
<td>58.07%</td>
<td>3.44%</td>
<td>28.18%</td>
</tr>
<tr>
<td>20</td>
<td>75.14%</td>
<td>58.73%</td>
<td>3.45%</td>
<td>28.31%</td>
</tr>
<tr>
<td>24</td>
<td>75.17%</td>
<td>58.69%</td>
<td>3.47%</td>
<td>28.41%</td>
</tr>
<tr>
<td>36</td>
<td>75.20%</td>
<td>58.96%</td>
<td>3.45%</td>
<td>28.50%</td>
</tr>
<tr>
<td>48</td>
<td>74.97%</td>
<td>58.81%</td>
<td>3.48%</td>
<td>28.27%</td>
</tr>
<tr>
<td>60</td>
<td>67.19%</td>
<td>58.53%</td>
<td>3.46%</td>
<td>27.95%</td>
</tr>
<tr>
<td>72</td>
<td>60.79%</td>
<td>58.53%</td>
<td>3.45%</td>
<td>27.53%</td>
</tr>
<tr>
<td>84</td>
<td>53.13%</td>
<td>57.80%</td>
<td>3.45%</td>
<td>27.22%</td>
</tr>
<tr>
<td>96</td>
<td>49.17%</td>
<td>57.70%</td>
<td>3.44%</td>
<td>26.48%</td>
</tr>
<tr>
<td>108</td>
<td>45.17%</td>
<td>57.70%</td>
<td>3.42%</td>
<td>26.73%</td>
</tr>
<tr>
<td>120</td>
<td>36.09%</td>
<td>57.16%</td>
<td>3.40%</td>
<td>26.24%</td>
</tr>
<tr>
<td>132</td>
<td>37.95%</td>
<td>56.72%</td>
<td>3.39%</td>
<td>25.81%</td>
</tr>
</tbody>
</table>

**Figure 7.20:** Efficiency results for Synthetic transactional with $\text{max}k = 5$

Figure 7.18 and Figure 7.20 illustrate the results obtained when the dataset was tested for efficiency.

Level 1 utilizes the parallel system well for up to 48 processors with efficiency levels to be approximately at 75%. Thus 75% of the total execution time was used to perform useful computation and around 25% was wasted as parallel overhead. When more than 48 processors are added the system’s efficiency decreases dramatically. It follows a line of descent form 75% (up to 48 processors) to 37.95% at 132 processors. The reason why this is observed is the same discussed in the previous paragraph for Level 1 speedup.
Although Level 2 starts at lower levels of efficiency (50.83% using 4 processors) compared to Level 1 (63.94% using 4 processors), it outweighs Level 1 somewhere beyond 72 processors. Level 2 was increasing at a higher rate than Level 1 for up to 72 processors. Due to the effects discussed in the speedup analysis section earlier, Level 1 starts to descent after 72 processors whereas Level 2 continues to improve in a linear fashion. When using 132 processors Level 1 achieves a speedup of 37.95% (62.05% parallel overhead) and Level 2 achieves a speedup of 56.72% (43.28 parallel overhead). Level 2 shows much greater scalability compared to Level 1 which shows excess fluctuations.

Level 3, as expected based on the discussion of the previous paragraph on speedup analysis, shows very poor performance. Efficiency levels show that only 3.58% (on average) out of the total execution time is spent in useful computation with a significant parallel overhead of 96.42%. Efficiency for Level 3 is a flat line that shows no positive trends as it appears to decrease very slowly (about 1% at a time) for more than 48 processors.

Opt_Level 3 amplifies efficiency levels to an average of 27.28% (from 3.58% at Level 3). Thus, 23.7% from the parallel overhead at Level 3 (96.42%) is now transformed in useful computation. Using 48 processors Level 3 spends 3.48% of the execution time in useful computation whereas Opt_Level 3 spends 28.27% in useful computation. Using 120 processors Level 3 spends 3.4% of the execution time in useful computation whereas Opt_Level 3 spends 26.24% in useful computation.
7.7.5 Comparison using Connect-4 dataset with $maxk = 5$ and $maxk = 12$

A very interesting observation can be seen from comparing the speedup and efficiency levels using the same dataset with different values for maximum $k$-MSU. The level shown in Figure 7.21 and Figure 7.22 is Opt_Level 3 for both $maxk = 5$ and $maxk = 12$. 
As it is illustrated in Figure 7.21, for $maxk = 5$ the rate at which speedup increases is not significant. It shows an increase of approximately 1 unit at a time (see Figure 7.4). There is an average of 6.06 of speedup achieved at that level.

As the value for $maxk$ grows larger, the rate at which speedup increases is remarkable (see Figure 7.9).

Using $maxk = 12$ there is an increase of approximately 4 units at a time (compared to 1 unit with $maxk = 5$) for up to 24 processors. Beyond 24 processors the rate increases to approximately 7 units at a time.

Furthermore, the system appears to operate more efficiently for larger $k$-MSUs.

For $maxk = 12$, an average of 73.04% of the execution is spent in useful computation (see Figure 7.10) whereas only 12.41% of the execution time is used in useful computation $maxk = 5$ (see Figure 7.6).

This shows that for small values of $k$, it is better to operate at higher levels of work generation. The tasks are already small in size thus any further division of them may harm efficiency as performance will decrease due to high communication overheads.

In other words creation of very small sub-tasks will increase the cost of the time taken for run-spawn-return cycle for those tasks.

As $k$ increases, performance may benefit from operating at lower levels of work generation. The tasks in such cases are relatively large and thus dividing them into smaller pieces may improve performance as the sub-search-to-processor ratio will increase. Nevertheless, the use of sub-division should be considered in moderation in order to avoid further division of tasks that are already small enough, resulting to the same problem as with small values of $k$. 

7.8 Summary

This chapter has presented the performance evaluation of parallel SUDA2 prior to and following the application of optimizations. Performance metrics, sources of overheads associated with parallelism as well as scalability issues were also presented.

Several experiments were performed on different datasets. The experiments were restricted to relatively small sizes of maximum size for k-MSUs in order to execute in reasonable time scales. The optimized parallel SUDA2 proved reasonably efficient. In every case the performance gains at a specific optimized level are much greater than the performance of that level prior to the optimization. For some cases it shows even greater performance gains compared to all levels.

The main conclusion is that for datasets that contains a small number of MSUs, it is better not to descend further than level 2 work generation. This is because the tasks are already small enough and breaking them down to even smaller sub-tasks, simply results in greater communication costs. As we move to datasets that contain larger size MSUs lower levels of work generation may increase performance. However descending down the tree to lower levels means very small tasks are likely to be encountered. Such tasks should not be divided any further. The further we descend down the recursion tree, the more selective task generation must be as the spawn-run-return cost overhead may surpass the benefit of having smaller sub-searches and high sub-search-to-processor ratio.

Furthermore the nature of the algorithm, and the optimization techniques being applied, seem to be more effective with datasets that cause the work to be accumulated towards the middle of the process. This is an issue to be considered for a SUDA3 implementation as there are datasets such as PUMS that do not follow this pattern.
Chapter 8

Conclusions

8.1 Summary and Findings

The release of data for statistical and other purposes has been shown to be potentially risky in terms of identification of certain individuals, by disclosing information that should not have been disclosed. An effective solution named SUDA2 has been developed that can identify combinations of attributes that can potentially disclose information. Whilst SUDA2 is a major improvement over its predecessor, nonetheless there are datasets that may require higher performance. Hence, the focus of this work has been the investigation of the parallel version of SUDA2.

Several algorithms for Mining Frequent Itemsets were presented in order to show trends towards performance gains. Among the different methods for improving performance, parallel algorithms seem to be the most effective. Furthermore similar ideas have been studied such as $k$-anonymity and Infrequent Itemset Mining. However none of them addresses directly the problem that SUDA2 focuses. Furthermore the steps towards the algorithm design have been shown as well as several new properties that can effectively prune the search space.
A parallel version of the sequential algorithm SUDA2 was presented; an optimization scheme for load balancing based on different size of parallel task generation has been proposed and tested in order to investigate the performance gain and usefulness of parallel algorithms.

The program was written in C++ programming language and implemented using the message-passing scheme in MPI.

The objectives of the parallel system design were scalability and efficiency. The reasons why these characteristics form the objective and evaluation of our work will be discussed separately in the following sections.

### 8.1.1 Scalability

Scalability was discussed extensively in the previous chapter. As already mentioned, a system is said to be scalable if it can maintain constant efficiency while the problem size increases.

The main reason that a scalable system is desirable is cost and effort reduction. A system that does comply with the scalability requirement will eventually become hard and costly to update and maintain. In particular the number of sub-searches when mining for MSUs can dramatically increase while descending down the recursion tree. Thus a scalable system is desirable. The nature of the problem does not have room for an algorithm that is not scalable. Imagine having to change the algorithm every time there is a different dataset or parameters to be used. This would be a hassle, especially for the programmer as well as for the person who will provide the financial support for this amendment.

The need for throughout and clear understanding of the algorithm, libraries, and architecture to be used is required in order to design a system that establishes scalability as its main characteristic.

Our parallel system has been proved to be relatively scalable thus able to accommodate most problems effectively in order to maintain constant efficiency.
Figure 8.1 illustrates the scalability of the parallel system when comparing Level 3 and Opt_Level 3 using the Connect-4 dataset with $\maxk = 5$. Where Level 3 demonstrates an increment of the rate of around 0.06 units at a time (as more processors are added), Opt_Level 3 demonstrates an increment of almost 1 unit a time (see Figure 7.4). Speedup appears to be increasing in a linear fashion proportional to the number of processors added.

Figure 8.2: Speedup comparison for Synthetic transactional dataset with $\maxk = 5$
Furthermore, Figure 8.2 illustrates the scalability of the parallel system when comparing Level 3 and Opt_Level 3 using the Synthetic transactional dataset with $\text{max} k = 5$.

Again the system demonstrates great scalability for even more than 140+ processors. Level 3 demonstrates an increment of 0.04 – 0.40 units at a time as more processors are added. Opt_Level 3 demonstrates an increment of 1.15 – 3 units at a time increasing in proportion to the processors added (see Figure 7.18).

### 8.1.2 Efficiency

Several experiments were performed using different data and parameters in order to evaluate the performance of parallel SUDA2. Performance can be captured by analyzing factors such as computation and communication. As the ratio between computation and communication increases, performance increases as well. Furthermore computation and communication can be affected by a number of parameters such as:

- Number of transactions
- Number of items in each transaction
- Size of MSUs
- Number of MSUs

We have used a variety of datasets of different sizes as well as different sizes for maximum size of MSUs to be analyzed. Optimizations were applied at specific levels of work generation resulting in a reasonable efficient parallel system. In every case the performance gains at a specific optimized level was much better than the performance of that level prior to the optimization. Figure 8.3 illustrates the results obtain when comparing efficiency obtained at Level 3 and Opt_Level 3 using the Connect-4 dataset with $\text{max} k = 12$. Initially the algorithm at Level 3 spends on average 52% of execution time doing useful computation. After the optimization is used, efficiency boosts to 73.04% at
Opt_Level 3. Thus a total increase of around 21% is gained when the optimization takes effect.

Figure 8.4 illustrates another case where efficiency is captured by comparing Level 3 and Opt_Level 3 using the Synthetic transactional dataset with $\text{max}_k = 5$. A significant improvement is observed in this case as efficiency is increased by a total of around 24%.

**Figure 8.3:** Efficiency comparison for Connect-4 dataset with $\text{max}_k = 12$

**Figure 8.4:** Efficiency comparison for Synthetic transactional dataset with $\text{max}_k = 5$
Level 3 spends on average only 3.58% in useful computation whereas Opt_Level 3 spends on average 27.28% performing useful computation.

We have observed that as the dataset as well as the size of MSUs grows larger, efficiency is higher (see Figure 7.22). When experiments were run using the Connect-4 dataset, $maxk = 5$ spends on average 12.41% of the execution time in useful computation. For $maxk = 12$ efficiency levels were significantly improved as they have reached an average of 73.04%.

In general the greater the number of MSUs to be detected the more efficient the optimized algorithm is, at that particular level.

It should be mentioned that in the case of the PUMS dataset the efficiency achieved was poor due to the unexpected (due to the nature of the algorithm) distribution of work. SUDA2 uses a heuristic referred to as middle-out, where the middle part of the recursive tree is selected to be processed first. This is based on the assumption that work is accumulated towards the middle of the process [36]. PUMS is a dataset that does not follow this characteristic as it demonstrates a skewed nature [36]. This should be considered when future enhancements are to be employed in the algorithm. A different method has to be discovered to perform a wiser work distribution. This is further explained at a later section in future work.

### 8.2 Future Work

#### 8.2.1 SUDA2-specific load balancing enhancements

As already discussed SUDA2 employs a middle-out work allocation scheme which basically starts from the middle of the search tree allocating work while moving outwards. This idea was inspired from the observation that most of the work tends to be accumulated towards the middle of the tree [36]. However this is not always the case. There is a possibility that the shape of the tree may vary between different datasets (such as the case with PUMS dataset). In addition even if the work accumulates in the middle, there is still the possibility that some anchors with long
sub-searches may be located towards the right or left hand side, away from the middle. Either one or the other case may cause some processors to remain idle until others have completed their work.

Adaptation of new ways that will produce an enhanced scheme of work allocation is one area that requires further investigation. Certain methods can be developed that apply specifically in SUDA2 such as Task Sorting, Job Batching, and Dynamic Level I Work Generation.

**Task Sorting:**
Unfortunately the amount of computation each anchor carries with it, is unknown during compilation time. This method must be able to predict at run time the amount of work that each anchor encloses. Then a sorting method (such as quick sort or heap sort) can facilitate in classifying the anchors in descending order based on the amount of computation each one requires. This will enable SUDA2 to distribute large tasks first, leaving smaller ones at the end. In this way the parallel system will be utilized more efficiently and more effectively as the processors will be kept busy at most times. In other words, large tasks will utilize completely the system for the longest part of the process. Towards the end of the process, even if there are not enough tasks to keep the system balanced, the negative effect will be negligible since the tasks at that point will be very small and will finish very quickly.

**Job Batching:**
Another way to deal with and reduce processors’ idle time is by grouping multiple jobs together and send them to be processed as one job. This requires estimation of certain small jobs by using timers to calculate the time spent in spawn-run-return cycle. Small jobs that their spawn-run-return cycle outweighs the time needed for computation, can be grouped together and delivered to a single processor. This will not only reduce communication overheads but also will increase the processor availability.

**Dynamic Level I Work Generation:**
The level of work generation is currently determined in a static fashion. The user provides SUDA2 with a parameter indicating the level at which they wish the algorithm to descend and generate work. This method requires the user to be
familiar with the input’s nature and also be able to estimate which level is the most appropriate to be supplied. However this is impossible to be true for all the users of the system. Thus a method is required that will use an estimation formula (supplied with the number of transactions, number of columns, number of processors, and $max_k$) in order to determine dynamically which level of work generation is the most appropriate for each particular case. As explained in Chapter 7, for some small values of $max_k$, producing finer granularity has resulted in hurting performance instead of amplifying it. Furthermore, for some large values of $max_k$, descending no further than Level 1 it can improve performance for a small number of processors. However as the number of processors expands performance levels drop as granularity is too coarse to utilize the system effectively.

It appears that a method that will be able to select the appropriate work generation level dynamically will reduce the processors’ idle time and facilitate load balancing.

### 8.2.2 Dynamic load balancing schemes

The load balancing enhancements described in the previous section refer to optimizations that apply to SUDA2 in particular. There exists some commonly used dynamic load balancing schemes that SUDA2 can employ and improve performance. These schemes are known as *asynchronous round robin, global round robin*, and *random polling* [19].

**Asynchronous Round Robin:**

In asynchronous round robin a distinct variable is associated with each processor indicating a “donor processor”. In case a processor finishes its work early, it uses the value of that variable in order to receive work from the donor processor. This value is incremented by one each time a work request is sent. The initial value of the variable is the value of its local processor + 1 (i.e. the one after the local processor). The value of a specific variable is independent of the values of other variables associated with different processors. Care must be taken whenever two distinct processors request work from the same donor.
Global Round Robin:
Global round robin is similar to the asynchronous round robin in the way that both keep a variable that indicates the donor processor. However, the main difference is that global round robin uses a single variable, available to all processors instead of one for each processor. This variable can be stored in a globally accessible place, in a share-address-space architecture or at a specific processor in message-passing. The value of the variable is incremented by one before responding to the next request. Whenever a processor runs out of work, a request can be send to the donor processor using the value of that variable. Furthermore, this scheme ensures an even distribution of work requests.

Random Polling:
This scheme is different from the ones previously presented. No variables are used in order to keep track of the next donor. Whenever a processor becomes idle, a request is sent to a randomly selected processor which will be the donor. There is equal probability for each processor to be selected as a donor. The scheme must ensure even distribution of work requests.

8.2.3 Producing dynamic work generation
The optimization method introduced and being used in parallel SUDA2, still operates on static work generation. It descends down to the appropriate level and each time, produces work generation only in the middle 50 percent of the tree. This was applied in order to reduce the communication costs caused by the generation of jobs that are too small and thus not worth sending them to a different unit for processing (due to high cost for spawn-run-return cycle).

It would be much better if a dynamic work generation scheme is introduced. Such a scheme must be able produce a desirable number of nodes for processing according to the number of nodes at each level $l$ (such as 50 percent of the middle nodes for any individual level). This could work as a step function which will eventually generate only work that it is worth to be sent for processing on a different unit.
In fact we have experimented with this idea by creating such a step function. Figure 8.5 illustrates this method.

While descending down the tree, the separator chooses only the middle part of each level to be generated.

We have tried this experiment using the Connect-4 dataset with $maxk = 12$ and $work\ generation\ level = 3$. Not only have the obtained results showed better performance than the optimized version at Level 3, but also the performance is even better than the one at Level 2. That is, using this step-function optimization technique we managed to make parallel SUDA2 to obtain better performance than before. The algorithm runs faster for all number of processors thus increasing speedup levels. Furthermore greater levels of efficiency were achieved, meaning that more execution time than before is used for computation avoiding some fluctuations observed at Level 2. Figure 8.6 presents a table including execution time between the normal Level 2 work generation, the optimized Level 3, and the optimized Level
3 with step function. The Level 3 optimization with step function appears to outperform all other previous performance achievements.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Level 2 time (s)</th>
<th>Opt_Level 3 time (s)</th>
<th>Opt_Level 3 with step function time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20439</td>
<td>20439</td>
<td>20439</td>
</tr>
<tr>
<td>4</td>
<td>8182</td>
<td>8374</td>
<td>8097</td>
</tr>
<tr>
<td>8</td>
<td>3568</td>
<td>3633</td>
<td>3546</td>
</tr>
<tr>
<td>12</td>
<td>2276</td>
<td>2325</td>
<td>2285</td>
</tr>
<tr>
<td>16</td>
<td>1674</td>
<td>1714</td>
<td>1663</td>
</tr>
<tr>
<td>20</td>
<td>1341</td>
<td>1366</td>
<td>1339</td>
</tr>
<tr>
<td>24</td>
<td>1134</td>
<td>1148</td>
<td>1125</td>
</tr>
<tr>
<td>36</td>
<td>792</td>
<td>793</td>
<td>778</td>
</tr>
<tr>
<td>48</td>
<td>612</td>
<td>628</td>
<td>610</td>
</tr>
<tr>
<td>60</td>
<td>538</td>
<td>527</td>
<td>519</td>
</tr>
<tr>
<td>72</td>
<td>445</td>
<td>455</td>
<td>443</td>
</tr>
<tr>
<td>84</td>
<td>395</td>
<td>402</td>
<td>392</td>
</tr>
<tr>
<td>96</td>
<td>370</td>
<td>371</td>
<td>359</td>
</tr>
<tr>
<td>108</td>
<td>338</td>
<td>341</td>
<td>335</td>
</tr>
<tr>
<td>120</td>
<td>313</td>
<td>320</td>
<td>316</td>
</tr>
<tr>
<td>132</td>
<td>299</td>
<td>307</td>
<td>296</td>
</tr>
</tbody>
</table>

Figure 8.6: Execution time between different work generation levels

While this experiment demonstrates better performance results, it cannot be used as a proof to jump to conclusions as it is the only one performed. Further experiments will be beneficial in order for the construction of an optimal step function. It may be worthwhile to adjust the work generation at each level.

8.2.4 Performance analysis

Several experiments have been carried out in this project for analyzing and assessing performance. Performance is a function of computation and communication. Especially in parallel mining for MSUs, these two factors depend heavily on the number of items, number of transactions, number of MSUs, and the size of each MSU itemset. Further experiments will be useful in order to examine the effect that all these parameters may have upon the performance of the parallel system. Although such a function that would combine all these factors will not be
straightforward to approximate, it is possible to investigate the effect of each of these parameters separately.

**Number of transactions and number of items:**
A function that combines the number of transactions in the dataset and the number of items on each transaction it is possible to be produced. An increase in the number of transactions not only means the existence of more possible MSUs in the dataset, but also means that additional scans are required. An increase in the number of items on each transaction implies more combinations to be tested for uniqueness. Thus a change on these parameters imposes a direct effect on computation as well as communication. Further experiments that show trends when changes on these parameters occur, would enable the algorithm to make a better selection on the optimal number of tasks a particular process need to be sub-divided and which granularity level is the most appropriate to be applied.

**Size of $k$-MSU itemset and number of MSUs:**
Another factor that directly affects the performance of the parallel system is the maximum size of $k$-MSU. An increase in the value of $k$ implies testing for larger combinations of itemsets for uniqueness. Larger combinations (i.e. larger itemsets) imply more MSUs to be discovered as there will be more items that can produce unique combinations. The more MSUs to be discovered the higher the amount of computation and communication to be performed is required. Additional experiments are required in order to identify trends that will enable the algorithm to make the correct decisions concerning the level at which work is preferred to be generated. Furthermore, this will facilitate the optimization technique presented in this dissertation, as better adjustments, on the percentage of work to be generated at each level, will be possible (instead of the fixed 50 percent used).

The values for $max_k$ (which determine the maximum size of MSUs) used to run experiments in this dissertation were relatively small to allow reasonable execution times, especially for small number of processors.

Further tests using larger values for $max_k$ are required in order to identify certain trends that the process follows. Such tests will require several days to run. In
addition the algorithm can be applied to a parallel architecture that accommodates a larger number of processors. This can be used to strengthen the argument that the larger the value of $maxk$ (i.e. more MSUs) the more the benefits gained by producing work at lower levels of granularity (such as Level 4, 5, 6…etc.).

8.2.5 Experiment with message size reduction

Based on the results obtained from the experiments, it has been observed that communication overhead is raised when messages are too large (i.e. large itemsets needs to be transferred). Methods that can result in message size reduction will probably facilitate performance. The way to do this would be to compact the message using some kind of compression algorithm at the source of the message, and decompression of the message at the destination. A method for accomplishing this was developed and tested by [42]. A message passing library was implemented, namely cMPI, which essentially compresses the message on the fly, achieving significant reduction in the total execution time. Unfortunately the library currently provides linking only to FORTRAN applications. An extension of this library that would include an interface with C/C++ applications would be a good candidate for experimentation as the total execution time for SUDA2 can be decreased if the overhead created by communicating messages is reduced.

This library exploits the TCP protocol to create a TCP connection between two communicating MPI processes. A message thread is generated for each process in order to handle send/receive operations from all communication channels. Furthermore, this thread is responsible for compressing the message if necessary, including a flag indicating whether compression was applied or not. This flag is critical in order for the receiver to decompress the message suitably. When a send operation is initiated in MPI, the data type of the message must be specified. Based on the data type, an appropriate compression technique is applied. This algorithm utilizes a prediction function [42] in order to estimate message entries based on earlier entries. The difference between the true value and the predicted on is then used in order to save bits. A hash function [42] is then used for compressing and
decompressing messages. Experiments demonstrate that this compression method facilitated the achievements of overall speedups of around 3% [42].

8.2.6 Experiment with reduction in database scans

Although this algorithm is designed to work with microdata, which are usually smaller compared to datasets used in frequent pattern mining, it is possible to achieve some performance gains by seeking database scan reduction. Two existing techniques appear to reduce database scans effectively. These are the creation of a data structure such as an **FP-tree** (to fit the dataset in an efficient way for searching) and representing the data using **vertical data representation**.

**FP-tree:**

In a parallel system such as the one used to run the experiments in this dissertation, each processor has a copy of the initial dataset, stored on its local memory for fast access. Although searching for MSUs achieves good performance gains, the algorithm may still suffer from the non-trivial cost of requiring multiple scans through the dataset (especially searching for long patterns) [7, 9]. Chapter 2 described a data structure referred to as FP-tree (used by the FP-growth algorithm) which is used in order to reduce database scans.

The tree essentially is an extended prefix-tree [7] that stores quantitative information about unique patterns. The tree will only consist of nodes for length-1 items while any further \(k\)-MSUs discovered will just increment counters stored in those nodes. The nodes will be arranged in such a way in the tree that infrequent nodes will have better chances of sharing nodes than frequent ones.

Furthermore, since FP-growth and SUDA2 are of similar nature (they are both divide-and-conquer algorithms) [10, 12], a method similar to the one employed by FP-growth, can be adapted for searching (see section 2.3). Such a method will check only the conditional pattern base of the tree and construct its corresponding conditional FP-trees (see section 2.3). Mining can then proceed by scanning the conditional FP-trees.
Employing FP-trees in SUDA2, the master processor could construct the tree and broadcast it to every worker processor. In this case workers will operate on this structure instead of the whole dataset.

This process will considerably reduce the dataset scans and it is therefore worthwhile to be developed and tested. In fact, according to [7], for Connect-4 dataset the compression ratio could be over 100.

**Vertical Data Representation:**

A different method for representing the data has been proven to be more efficient in database scanning [7, 11]. This method is known as vertical data representation and is currently being used by the MAFIA algorithm presented in Chapter 2.

SUDA2 uses the horizontal data representation which essentially organizes the dataset as a set of rows, each one representing a transaction (see Figure 2.3).

On the other hand, vertical data representation associates with each item a set of transaction identifiers. The vertical representation version of the dataset in Figure 2.3 is illustrated in Figure 8.7.

<table>
<thead>
<tr>
<th>Itemset</th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>{T100, T400, T500, T700, T800, T900}</td>
</tr>
<tr>
<td>I2</td>
<td>{T100, T200, T300, T400, T600, T800, T900}</td>
</tr>
<tr>
<td>I3</td>
<td>{T300, T500, T600, T700, T800, T900}</td>
</tr>
<tr>
<td>I4</td>
<td>{T200, T400}</td>
</tr>
<tr>
<td>I5</td>
<td>{T100, T800}</td>
</tr>
</tbody>
</table>

**Figure 8.7:** The vertical data representation version of the dataset in Figure 2.3

Using this data format, SUDA2 can take advantage of a technique named as *diffset propagation* [39]. Diffset propagation essentially takes the difference between two transaction sets. For instance, in Figure 8.7 we have \{I1\} = \{T100, T400, T500, T700, T800, T900\} and \{I1, I2\} = \{T100, T400, T800, T900\}. The diffset between these two sets is \textit{diffset}(\{I1, I2\}, \{I1\}) = \{T500, T700\}. Thus, instead of recording the four TIDs obtained from \{I1\} \cap \{I2\}, the diffset can be used to record just the two TIDs indicating the difference between the two sets.

Experiments have demonstrated that in many cases vertical data representation achieves greater performance gains over the horizontal one [7, 39].
8.2.7 Implementation of SUDA2 using POSIX Threads

“Forking” a new process in order to run on a different processor, in a multiprocessor environment, it is very expensive. Whenever a new process is spawned, the system has to initialize a new virtual memory space and environment for that process. Furthermore, whenever a context switch occurs (either for I/O or communication), the process has to save its state, which includes information such as process instructions, data, open files, user and group id, etc. A more efficient way to achieve concurrency is realized using threads. Threads (also known as lightweight processes), are simply a way to split a program into quasi-concurrent tasks [40]. The benefits of threads lie behind the fact that a group of threads is associated with a specific process allowing them to share the memory of that process. Since threads share the same memory between them, the overhead associated with a context switch or spawning, is much less compared to processes. There is no need to save any data or process related information. They only need to save information concerning the instruction pointer, register values, and thread specific data. POSIX threads or Pthreads, are standard libraries defining APIs for creating and manipulating threads in UNIX [40, 41].

In a shared-address-space or a MTA (Multi Threaded Architecture) architecture (see section 6.1), Pthreads can be adapted by SUDA2, to implement parallelism. This can offer additional performance gains for several reasons. First, based on the fact that all the threads in a process share the same memory, inter-thread communication will be more efficient than inter-process communication [40]. Second, asynchronous event handling is possible, where tasks that handle events of undetermined frequency and duration can be interleaved [40]. Finally, threads enable tasks that are more important than others to be scheduled to run first during run-time [41]. This could be used in order to facilitate load balancing in SUDA2. The system could assign priority to tasks during run-time and decide which tasks need to be processed before others. Since computation of each task is unknown during compilation time this feature will overcome the problem of having large tasks to wait for small ones to finish execution allowing for a high sub-search-to-processor ratio at all times.
Implementing a multithreaded program is not an easy task as care must be taken in order to avoid potential errors such as *race conditions, deadlocks, and priority failures* [41]. Race conditions occur when multiple threads attempt to read and write at the same memory location at the same time. Lock mechanisms such as semaphores, must be used in order to ensure that only one thread at a time has access to a critical section of the code. A deadlock arises when a thread $A$ is waiting to use a resource used by another thread $B$, while $B$, at the same time, is waiting to use a resource currently used by $A$. Consequently, threads $A$ and $B$ may wait forever. Certain algorithms exist that can be used in order to avoid such cases [41]. Priority failures refer to cases were the sequence in which threads are executed is not performed within the desired time scales. Care must be taken to avoid such cases. An implementation of a SUDA2 version that takes advantage of Pthreads will be a good idea to experiment.

Using Pthreads with MPI requires each process to create and manage $n$ threads ($n$ varies between different architectures and applications), where $n$ utilizes the available processors as efficiently as possible.

### 8.2.8 Overlapping Computation and Communication

Another way to improve the performance of the parallel system would be to further overlap computation and communication. As soon as a worker finishes working on its allocated tasks a message is sent to the master processors to notify it, and consequently request for more work if available. This idea causes the worker node to become idle until a new job is received. A heuristic method could determine whether a specific worker processor will finish the work being allocated and send a message and request for more work before it actually finishes. By the time the worker node finishes processing, new jobs are readily available for processing thus saving idle time.

Another way to achieve overlapping is by sending one extra job at a worker. As soon as the worker finishes the first job, it will start processing the second one while a request for an extra job can be initiated simultaneously.
In order to overlap computation and communication in the message-passing paradigm, non-blocking message passing primitives must be used. This will allow communications and computations to proceed concurrently. In addition the developer must ensure that the underline architecture supports these primitives.

Furthermore, achieving overlap in a shared-address-space architecture requires pre-fetching hardware [19, 20]. Pre-fetching will initiate access to a memory address anticipated that will be needed in the near future.

Efficient overlapping of computation and communication can reduce significantly overheads associated with communication [19].

### 8.2.9 Comparison with other infrastructures

A very interesting investigation would be to compare our proposed system with a parallel machine or with a network consisting of a greater number of processors than the ones used in this case. The plots obtained from experiments hint that lower levels of work generation may overtake higher levels somewhere beyond the 132 processors being used in these experiments. In addition a comparison between cost and performance could be performed.

Another good idea would be to compare parallel SUDA2 with another software design to solve exactly the same problem. Unfortunately this is not possible at the moment as we are unaware of any other application that has been designed for this purpose.

### 8.3 Summary

Summarizing what we have studied, this dissertation has successfully described and evaluated a parallel prototype for mining Minimal Sample Uniques as well as identified and implemented further optimization areas. This prototype is scalable and efficient.

Furthermore, a number of areas of further investigation have been identified.
Reference List


[9] Jiawei Han, Y., Pei J. and Yin Y. (2000), Mining Frequent Patterns without Candidate Generation. In *Proceedings of the ACM SIGMOD Conference, Dallas, Texas*, pages 1-12.


