Parallelization of Formal Concept Analysis Algorithms

Gamhewage Nuwan Kodagoda

A thesis submitted in partial fulfilment of the requirements of Sheffield Hallam University for the degree of Doctor of Philosophy

November 2018

Abstract

Formal Concept Analysis provides the mathematical notations for representing concepts and concept hierarchies making use of order and lattice theory. This has now been used in numerous applications which include software engineering, linguistics, sociology, information sciences, information technology, genetics, biology and in engineering. The algorithms derived from Kustenskov's CbO were found to provide the most efficient means of computing formal concepts in several research papers. In this thesis key enhancements to the original CbO algorithms are discussed in detail. The effects of these key features are presented in both isolation and combination. Eight different variations of the CbO algorithms highlighting the key features were compared in a level playing field by presenting them using the same notation and implementing them from the notation in the same way. The three main enhancements considered are the partial closure with incremental closure of intents, inherited canonicity test failures and using a combined depth first and breadth first search. The algorithms were implemented in an un-optimized way to focus on the comparison on the algorithms themselves and not on any efficiencies provided by optimizing code.

One of the findings were that there is a significant performance improvement when partial closure with incremental closure of intents is used in isolation. However there is no significant performance improvement when the combined depth and breadth first search or the inherited canonicity test failure feature is used in isolation. The inherited canonicity test failure needs to be combined with the combined depth and breadth first feature to obtain a performance increase. Combining all the three enhancements brought the best performance.

The main contribution of the thesis are the four new parallel In-Close3 algorithms. The shared memory algorithms Direct Parallel In-Close3, the Queue Parallel In-Close3 algorithm and the Distributed Memory In-Close3 algorithm showed significant potential. The shared memory algorithms were implemented using OpenMP and the distributed memory algorithm was implemented using MPI. All implementations were validated and showed scalability. Experiments were carried to test the features of the parallel algorithms and their implementations using the UK National Super Computer Archer and Colfax Clusters. The thesis presents the key parallelization strategies used and presents experimental results of the parallelization.

Acknowledgements

I am grateful to my research supervisor Prof. Simon Andrews of the Department of Computing, Faculty of Arts, Computing, Engineering and Sciences, Sheffield Hallam University for his guidance, untiring support and encouragement during this research.

I also wish to thank the management of SLIIT for sponsoring my PhD and providing me support to pursue my research work. I would like to specifically thank Prof Lalith Gamage, President/CEO of SLIIT and former Chairman of SLIIT Prof Sam Karunarathne for the support and encouragement they have extended to me during this period. I also wish to express my gratitude to the Deans of the Faculty of Computing, during this period, Prof Koliya Pulasinghe who is also my internal supervisor at SLIIT and Dr Malitha Wijesundara for their kind encouragement and support.

I wish to thank my parents for supporting me and guiding me to pursue an academic life, I am much indebted to them. My sister Neesha, for her continuous support and encouragement. She also put up with me and provided lodging during my annual visits to UK.

Finally I am much indebted to my wife Sudheera without whom it would not have been possible to complete this work.

Dedication

For my wife Sudheera, my daughters Sawanya and Savithi and my sister Neesha.

Publications

The following research articles were published based on the research work carried out.

Kodagoda, Nuwan, and Pulasinghe, Koliya (2016). "Comparision Between Features of CbO based Algorithms for Generating Formal Concepts." International Journal of Conceptual Structures and Smart Applications (IJCSSA) 4.1 (2016): 1-34.

Kodagoda, Nuwan, Andrews, Simon and Pulasinghe, Koliya (2017). A parallel version of the in-close algorithm. In: NCTM 2017 Proceedings of the 6th National Conference on Technology and Management (NCTM). IEEE, 1-5.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Introduction

There is a huge amount of raw data that is produced each day. Since this raw data cannot be used as it is, knowledge in form of structures or patterns needs to be extracted from the data. This extraction of knowledge from data is known as data mining, and data analysis is the term used for further analysis and transformation of the structures found (Poelmans, Elzinga, Viaene, & Dedene, 2010; Poelmans, Ignatov, Kuznetsov, & Dedene, 2013a; Wille, 2002).

Formal Concept Analysis (FCA) is a contemporary data mining and data analysis technique for object-attribute relational data. FCA can be used to generate a concept hierarchy from a collection of objects and their properties (Stumme, 2009). Philosophically a concept can be considered as a unit of thought (Wille, 2005). Concepts allow us to generalize real world or abstract ideas enabling higher level of thinking (Strahringer, Wille, & Wille, 2001). In FCA concepts are precisely defined mathematically and these concepts are known as formal concepts. The sub concept, super concept relationships between concepts found in a given domain can be represented in a concept hierarchy. Concept hierarchies are generally represented as trees, although a lattice structure is the preferred representation in FCA (Wille, 2005). Traditionally concept hierarchies are created by human experts in a given domain, there are obvious benefits in having an automated way of generated concept hierarchies (Alani et al., 2003; Uschold & Gruninger, 1996). Formal Concepts which are the elementary output generated by FCA algorithms can in turn be transformed to association rules using data mining algorithms. There are many applications of FCA in diverse fields including data mining, machine learning, knowledge management, semantic web, software engineering, chemistry and biology (Andrews, 2015; Poelmans et al., 2010; Poelmans, Ignatov, Kuznetsov, & Dedene, 2013b; Tilley, Cole, Becker, & Eklund, 2005).

In order to define the complexity of the algorithms that generate formal concepts a brief introduction of the terms are described in this chapter, these are formally covered in depth in Chapter 3 of the thesis. In Formal Concept Analysis, a formal context is defined as a set of structure $K = (G, M, I)$ where G and M are sets representing all the objects and attributes respectively for a given dataset. I represent a binary relationship between G and M. All the generated concepts of a context can be represented in a Lattice represented by L. Then the total number of concepts can be represented by $|L|$. If we consider $|G|$ to be the number of objects in a given dataset, then number of concepts is at most $2^{|G|}$. The time complexity of generating all concepts is in general a polynomial function with respect to the number of objects (Hitzler & Schärfe, 2009). For example, Kuznetsov's CbO algorithm has a time complexity of $O(|G|^2|M||L|)$ and is therefore computationally time consuming. Here $|M|$ represents the number of attributes in the given context (Kuznetsov & Obiedkov, 2002). This is significant for large datasets having millions of objects. Hence there is a need for faster algorithms that are able to tackle large datasets (Old & Priss, 2004).

Classical FCA algorithms for example Chein, NextClosure, Norris, Bordat were only capable of handling smaller datasets running upto only thousands of objects (Bordat, 1986; Chein, 1969; Ganter, 1984; Norris, 1978). There have been numerous attempts to build upon the classical FCA algorithms. Recent algorithms such as AddIntent, Berry, FCbO and the In-Close family of algorithms, improve upon the classical algorithms (Andrews, 2009, 2015; Berry, Bordat, & Sigayret, 2007; Outrata & Vychodil, 2012; van der Merwe, Obiedkov, & Kourie, 2004).

Almost all the FCA algorithms proposed in the literature to date are serial algorithms. Out of the few attempts to parallelize FCA algorithms, most of them have been theoretical prototypes (Fu & Foghlu, 2008; Fu & Nguifo, 2004; Kengue, Valtchev, & Djamegni, 2007; Xu, de Fréin, Robson, & Foghlú, 2012). The only notable parallelization effort have been Krajca's and Outrata's PCbO and PFCbO (Krajca, Outrata, & Vychodil, 2008, 2010b, 2010a; Outrata & Vychodil, 2012).

1.2 Research Questions

The fundamental research question this thesis addresses is: what is the best parallel solution for computing formal concepts? Is there a parallel solution that outperforms all other parallel and serial solutions? The most important feature of performance is the speed of computation, but factors such as scalability should also be considered. It may be that the 'best' solution involves some compromise between speed, scalability and memory requirements. Clearly, an exhaustive comparison of all solutions is beyond the scope of this work, but a sufficient answer to the research question will be provided by defining and examining a representative subset of all solutions.

The fundamental research question will be answered by answering the following sub-questions:

1) What is the most appropriate existing serial algorithm to choose for parallelization? This may simply be the most efficient serial algorithm, but it is possible that some algorithms are not suitable for parallelization and appropriateness should also consider scalability and memory requirements.

2) What are the options for parallelizing the chosen serial algorithm (e.g. shared memory, distributed solutions) and which options may be the best in terms of speed and scalability?

3) How do parallel versions of the chosen serial algorithm compare with each other and with existing parallel solutions?

These three questions are answered in the thesis and a guide on where these questions are addressed are given in Section 1.3

1.3 Objectives and Structure of the Thesis

To answer the first sub question, what is the most appropriate existing serial algorithms to choose for parallelization the thesis presents an analytical and empirical comparison of contemporary serial formal concept analysis (FCA) algorithm with the aim of selecting the fastest serial FCA algorithm. The selected algorithm was compared with the rest of the algorithms to check if it had any specific disadvantage for parallelization. Next the selected algorithm was parallelized using both shared memory and distributed memory parallelization strategies. Finally, the different variations of the parallel implementations were compared to determine the best parallel FCA algorithm.

The second sub question on parallelizing the chosen serial algorithm is answered in Chapter 5 and Chapter 6. These two chapters present detail descriptions of new, shared memory and distributed memory parallel FCA algorithms. The parallel algorithms presented in this thesis are parallel variants of the fastest serial FCA algorithm. Several researches have concluded through empirical testing that the CbO family of algorithms provides the best performance (Andrews, 2009, 2011, 2015, Krajca et al., 2008, 2010a; Outrata & Vychodil, 2012; Strok & Neznanov, 2010). The first sub question on identifying the best serial algorithm is presented in detail in Chapter 4. This chapter describes the comparison of eight different variants of the CbO algorithm in a level playing field. The algorithms, which highlight the key features found in contemporary CbO based algorithms, are presented using the same pseudocode notation, and implemented from the pseudocode notation in the same way. The fastest serial FCA algorithm selected by analytical and empirical comparison was selected as the algorithm to be parallelised. The final sub question on the comparison of the parallel versions is addressed in Chapter 5 and Chapter 6. Here the implementations of the new parallel algorithms are compared with their serial counterparts and other parallel implementations.

1.3.1 Methodology

Details of the methodology used in this research is presented in Chapter 2. This include details of how key serial FCA algorithms were compared, and the evaluation and comparison of the new parallel algorithms presented in this thesis.

1.3.2 Formal Concept Analysis

A background to formal concept analysis is presented in Chapter 3. Many serial algorithms generate all the formal concepts of a given context. The limitations of serial formal concept analysis algorithms are presented in Section 3.3. A brief summary of existing parallel algorithms is presented in Section 3.4. The rational for developing parallel formal concept analysis algorithms and the research work carried out is also presented in Chapter 3.

1.3.3 Comparison of existing serial algorithms

In Chapter 4, existing serial FCA algorithms that generate formal concepts are compared analytically and empirically. The algorithms and their implementations are compared in a level playing field. Here a similar notation is used to represent each algorithm and their implementations. Specifically eight variations of the CbO based algorithms are compared in detail. Contemporary CbO based family of algorithms performed better than other algorithm according to several researchers. The eight variations highlight the three key features that are prominent in recently developed CbO algorithms (FCbO, In-Close, In-Close II, In-Close III). The fastest serial FCA algorithm found in this analysis was selected as the candidate algorithm for parallelization efforts. The selected algorithm was also checked to ensure that it was suitable for parallelization. To validate that the fastest serial algorithm produces the best performance, two other serial algorithms that were considered were also parallelized and their results compared.

1.3.4 Shared Memory Parallel Algorithms

Modern computers are shared memory parallel machines with multiple cores. A program that fully utilizes the modern computer hardware has to be explicitly programmed using a shared memory programming framework. Chapter 5, presents three new shared memory FCA algorithms that parallelize the best serial FCA algorithm found in Chapter 4. These algorithms were implemented in C++ using the OpenMP shared memory framework. A detail empirical comparison of several additional parallel algorithms that are based on key serial algorithms presented in Chapter 4 is also made. Finally, this chapter presents optimization strategies used in the parallel implementation.

1.3.5 Distributed Memory Parallel Algorithms

Shared memory machines have limitations when it comes to scaling. Clusters that consist of several independent computers connected through a high bandwidth network are commonly used to build powerful parallel machines. Clusters, which are scalable parallel computers, are essentially distributed memory computing machines with each computer (node) in the cluster having its own private memory. Distributed parallel algorithms typically use message passing between computing nodes to coordinate parallel computation. A new distributed parallel FCA algorithm and its implementation is presented in Chapter 6. This parallel algorithm is also based on the fastest serial FCA algorithm selected from Chapter 4.

1.4 New FCA algorithms presented in this Thesis

The thesis presents seven new FCA algorithms (See Table 1.1). The major contribution of the thesis are the four new Parallel algorithms presented in Chapter 6 and Chapter 7. They are the Naïve Parallel In-Close3, Direct Parallel In-Close3, Queue Parallel In-Close3 and the Distributed Parallel In-Close3 algorithms. In addition, there are three new FCA algorithms presented in Chapter 4. These three new serial FCA algorithms helped in identifying the impact of the three enhancements used in modern CbO based FCA algorithms. The three algorithms are the CbO Full Closure Combined Depth First and Breadth First Search Algorithm (CbO-FC-DBF), CbO Full Closure Inherited Canonicity Test Failure and Depth First Search Algorithm (CbO-FC-ICF-DF) and the CbO Partial Closure Inherited Canonicity Test Failure and Depth First Search Algorithm (CbO-PC-ICF-DF). The algorithms are summarized in *Table 1.1*, in the order they are presented in the thesis.

Table 1.1, List of new algorithms presented in the thesis

New Algorithm	**Type of Algorithm**	**Presented In**
CbO-FC-DBF	Serial	Figure 4.6
CbO-FC-ICF-DF	Serial	Figure 4.8
CbO-PC-ICF-DF	Serial	Figure 4.11
Naïve Parallel In-Close3	Parallel Shared Memory	Figure 5.1
Direct Parallel In-Close3	Parallel Shared Memory	Figure 5.8
Queue Parallel In-Close3	Parallel Shared Memory	Figure 5.9
Distributed Parallel In-Close3	Parallel Distributed Memory	Figure 6.6

2. METHODOLOGY

2.1 Introduction

This research involved developing new algorithms and optimizing existing algorithms that can generate formal concepts for large data sets. This required a systematic methodology in initially identifying the best serial candidate algorithms which exist. The selected algorithms were taken forward into making parallel versions. At the end of the research, the performance of these newly modified algorithms, were systematically compared with existing algorithms.

The comparison of the efficiencies of different algorithms needed to be conducted at both the theoretical and practical level (Balakirsky & Kramer, 2004).

To theoretically compare several algorithms the asymptotic analysis of each algorithm needs to be carried out separately. In essence a complexity function is estimated for a large input, the Big O notation, Big omega notation and the Big theta notation can be used for this purpose (Cormen, Leiserson, Rivest, & Stein, 2001). However, the algorithms compared in this thesis have the same theoretical complexity of $\mathrm{O}(|\mathrm{G}|^2|\mathrm{M}||\mathrm{L}|)$. Where $|G|$, $|M|$ and $|L|$ represents the number of objects, the number of attributes and the number of concepts respectively. Here G, M and L represents all the objects, attributes and the concepts in a given dataset. A detail complexity analysis of each of these recursive algorithms is a complex task and beyond the scope of this thesis. An analytical comparison of the different serial algorithms was carried out instead (See Section 4.6). The approach used in the analysis is based on the fundamental method used by Cormen (Cormen et al., 2001). The purpose of the analytical comparison was to compare the complexity of the algorithms considered. This theoretical analysis was adequate to supplement the experimental results.

To compare different algorithms practically, carefully designed experiments needed to be carried out to test the running time of actual programs that implement these algorithms (Balakirsky & Kramer, 2004).

The experimental research methodology was used in this research to practically compare the different algorithms considered and presented. This methodology is the basis of the scientific method where the researcher manipulates one or more variables, and controls and measures any changes in other variables (Kumary, 2005; Ross, Morrison, & Mahwah, 2004). Here a series of carefully controlled experiments were designed and conducted. Andrews, 2015; Krajca, Outrata, & Vychodil, 2010; Kuznetsov & Obiedkov, 2002; Outrata, 2015; Strok & Neznanov, 2010 have all used a similar approach to compare actual programs that implement a set of algorithms.

Here the dependent variable time was measured varying each one of the independent variables, the number of objects, the number of intents and the density of a context while keeping the other two independent variables fixed.

2.2 Method of Sampling

The basic technique used for measuring time in a computer system is similar to how a stop watch can be used to measure a running experiment (Lilja, 2005). A computer system has an internal counter that measures the number of clock ticks from the time the computer was switched on. To measure the time taken for an experiment to run, the researcher would have to use a library function that would return the number of clock ticks. The difference of two measurements one taken one at the beginning and the second at the end of the experiment is the number of clock ticks taken for the experiment. This can be then converted using another library function to a wall clock time (Lilja, 2005).

A program's execution time is non deterministic, hence the measurement needs to be taken multiple times. It is misleading to represent the summary of multiple readings using a single statistical value such as the mean or the median. Appropriate statistical rigor is needed to ensure that the measurements can be relied upon to infer a decision regarding the different algorithms that are measured. This includes identification and removal of outliers.

There are two types of errors that can happen when measuring the time taken to run a program (Lilja, 2005).

1. Experimental Mistake

2. Random Errors

Care was taken to avoid experimental mistakes by double checking the exact part of the code that needed to be measured was consistent across the different programs. The algorithms were implemented using common code blocks which ensured that all the programs considered were written in a similar level of complexity and represented the high-level logic of the algorithms that they represented.

Random errors on the other hand are completely non-deterministic and needs to be handled using a statistical approach. These could be due to the result random processes within the system at both hardware and software level.

Vladimirov had noted that the first run of a program is slower than its subsequent runs. In the experiments carried by Colfax the first three timing results were ignored to eliminate this known outlier (Vladimirov, Asai, & Karpusenko, 2015). In this research every experiment was measured 13 times, the first 3 sample reading were ignored as outliers. The remaining 10 sample reading was used to represent the timing of running the program.

Total Program Execution time = $T1 + T2 + T3$

Where

$T1$ = Time taken to read sample data set

$T2$ = Time taken to execute the algorithm

$T3$ = Time taken to store and display the results

Both $T1$ and $T3$ required Input/Output from secondary storage devices but was the same across all algorithms for a given dataset. To further eliminate potential random errors due to Input/Output only $T2$ was measured for all programs. Instrumented code was placed in the main program just before and after the function call to execute the complete algorithm.

The mean, median, Interquartile Range, Outlier Boundaries and 95% level confidence values were computed for each of the samples assuming the samples follow a T Distribution.

Table 2.1, Statistical Analysis carried out for one raw experimental result

Dataset : Mushroom - Algorithm : CbO.PC.DBF (In-Close 2)		
Sample	Concepts	Time
1	226921	0.9184910
2	226921	0.8537550
3	226921	0.8541490
4	226921	0.8537010
5	226921	0.8534310
6	226921	0.8540500
7	226921	0.8534830
8	226921	0.8534220
9	226921	0.8539590
10	226921	0.8536050
11	226921	0.8536480
12	226921	0.8539750
13	226921	0.8534100

mean	0.8536684
median	0.8536265
min	0.85341
max	0.85405
IQR	0.00067575
95% confidence error	0.000176484
Outlier Boundary 1 = (Q1 - 1.5 x IQR)	0.85276825
Outlier Boundary 2 = (Q3 – 1.5 x IQR)	0.85457025

Table 2.1 shows how statistical analysis was carried out for one single experimental reading. Raw readings are presented as timing in this Table. From the 95% confidence level from this sample we can conclude that the measurements in this specific instance is accurate to the first 3 decimal places. This specific sample was for the timing result $T2$ for running the program representing the algorithm CbO-PC-DBF (In-Close2). The program was run 13 times, the first three results were excluded from the statistical analysis as outliers. We can clearly see in the above example that first reading is an outlier.

The mean, median, inter quartile range, the outlier boundaries the minimum and maximum of the last ten values were computed. If the minimum or maximum values were outside the outlier boundaries such samples were eliminated and all statistical values were re-computed using the available data set. In the majority of the samples all the values were within the permissible boundaries. The 95% confidence level was computed assuming that the distribution followed a T-Distribution. This was because the sample size was less than 30 (Georges, Buytaert, & Eeckhout, 2007)

The 95% confidence levels are shown in Tables that represent real datasets. One can observe that the experimental data cluster around the mean. This is because the programs were compiled and run using the C language which doesn't require any runtime environment to execute the programs. In addition, the computer used for the experiment was a node in a cluster running a minimalistic Linux based operating system. The executable programs and the datasets were submitted to the computer using a login terminal node. The Colfax cluster which was used to run the system ensured that the computational nodes only ran the submitted program at a given point in time. This ensured that there were no side effects during the experiment due the execution of other background programs.

2.3 Statistical Significance of empirical results

2.3.1 Introduction

The one-way analysis of variance (one way Anova) and the t-Test for Two sample assuming unequal variances were used to show that the empirical results were statistically significant.

2.3.2 One-way analysis of variance

To demonstrate that the empirical timing results between the implemented algorithms were statistically significant Kruskal-Wallis one way analysis of variance (one way Anova) was used (Kruskal & Wallis, 1952). An example is provided to describe how this analysis was carried out. *Table 2.2* contains the actual experimental data of the 10 sample runs for the mushroom dataset using the eight serial implementations of the algorithms described in Chapter 4 of the thesis.

The null hypothesis H_0 assumes that means of experimental run time of all the implemented algorithms are the same. The alternative hypothesis is that there exists at least one mean which is different from another mean.

$$H_0: \mu_1 = \mu_2 = \mu_3 = \cdots = \mu_8$$

$$H_A: \exists\ i, j: \mu_i \neq \mu_j$$

Table 2.2, Empirical Timing (seconds) for each serial algorithm running the mushroom dataset

	CbO-FC-DF (CbO)	CbO-PC-DF (In-Close)	CbO-PC-DF (In-Close2)	CbO-FC-ICF-DBF (FCbO)	CbO-PC-ICF-DBF (In-Close3)	CbO-PC-ICF-DF	CbO-FC-ICF-DF	CbO-FC-DBF
	1.082110	0.865881	0.853701	0.301187	0.252722	0.460980	1.067110	1.041970
	1.020000	0.865915	0.853431	0.301344	0.252731	0.460640	1.067110	1.042270
	1.022090	0.865790	0.854050	0.301287	0.252843	0.461190	1.066770	1.042280
	1.019840	0.865446	0.853483	0.301485	0.252766	0.460650	1.066840	1.041910
	1.019860	0.865822	0.853422	0.301174	0.253228	0.465750	1.067080	1.042090
	1.020130	0.866199	0.853959	0.301353	0.253242	0.460570	1.066540	1.042130
	1.020020	0.865580	0.853605	0.301196	0.253294	0.461650	1.066630	1.041920
	1.019940	0.865664	0.853648	0.301207	0.253263	0.462210	1.066520	1.041860
	1.020170	0.865446	0.853975	0.301416	0.253274	0.460730	1.066440	1.041870
	1.019770	0.866199	0.853410	0.301313	0.253302	0.460500	1.066930	1.041910
Mean	1.026393	0.865794	0.853668	0.301296	0.253067	0.461487	1.066797	1.042021
Rank	6	5	4	2	1	3	8	7

The one-way Anova which is also known as the single factor analysis of variance was carried out using Microsoft Excel. The output generated from the analysis using Excel is shown in *Table 2.3* with the significance level set to 5%.

Table 2.3, One Way Analysis of variance (one way anova) for dataset in Table 2.2

SUMMARY

Groups	Count	Sum	Average	Variance
CbO-FC-DF	10	10.263930	1.026393	0.000383718
CbO-PC-DF	10	8.657942	0.865794	7.28848E-08
CbO-PC-DF	10	8.536684	0.853668	6.08649E-08
CbO-FC-ICF-DBF	10	3.012962	0.301296	1.11922E-08
CbO-PC-ICF-DBF	10	2.530665	0.253067	6.85845E-08
CbO-PC-ICF-DF	10	4.614870	0.461487	2.54153E-06
CbO-FC-ICF-DF	10	10.667970	1.066797	6.60456E-08
CbO-FC-DBF	10	10.420210	1.042021	2.57656E-08

ANOVA

Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups (B)	8.156074	7	1.165153365	24112.95822	1.2384E-118	2.139655512
Within Groups (W)	0.003479	72	4.83206E-05			
Total (T)	8.159553	79				

The Summary Section of *Table 2.3* calculates the sum, average and variance of the experiment of the dataset presented in *Table 2.2*.

In the ANOVA section of *Table 2.3* of the analysis the following definitions and formulae is used.

The sum of squares SS_j, SS_T, SS_W and SS_B are defined as follows.

$$SS_j = \sum_i (x_{ij} - \bar{x}_j)^2$$

$$SS_T = \sum_j \sum_i (x_{ij} - \bar{x})^2$$

$$SS_W = \sum_j \sum_i (x_{ij} - \bar{x}_j)^2$$

$$SS_B = \sum_J n_j (x_{ij} - \bar{x}_j)^2$$

Where n is the sample size of *jth* sample, $\bar{x}_j$ is the mean of the *j*th group sample and $\bar{x}$ is the mean of the entire sample.

n is defined as follows.

$$n = \sum_{j=1}^{k} n_j$$

Where k is the number of samples.

The degrees of freedom df_T, df_B and df_W are defined as follows.

$$df_T = n - 1$$

$$df_B = k - 1$$

$$df_W = \sum_{j=1}^{k} (n - 1) = n - k$$

The mean square is defined as follows.

$$MS = SS/df$$

The mean square MS_T, MS_B and MS_W are defined as follows.

$$MS_T = SS_T/df_T$$

$$MS_B = SS_B/df_B$$

$$MS_W = SS_W/df_W$$

The F value is defined as follows.

$$F = \frac{MS_B}{MS_W}$$

The P_{value} for the (right tailed) F probability distribution for two data can be computed using the $FDistribution()$ function which requires F, df_B and df_W as parameters.

$$P_{value} = FDistribution(F, df_B, df_W)$$

The F_{crit}value is the inverse of the (right tailed) F probability distribution and can be obtained using the $FInverse()$ function which requires α, df_B and df_W as parameters.

$$F_{crit} = FInverse(\alpha, df_B, df_W)$$

Here α is the significance level.

We can reject the null hypothesis H_0 if $F > F_{crit}$ and $P_{value} < \alpha$

2.3.3 t-Test for Two sample assuming unequal variances

To show that the empirical timing results obtained for the two algorithms which needed to be compared has statistical significance the t-Test for two sample assuming unequal variances (Boslaugh, 2012) was used. In most cases the comparison was between the two algorithms that produced the best results. It was also used to compare the newly developed parallel algorithms with existing parallel algorithms.

The null hypothesis *H₀* assumes that the means of experimental run time of both the implemented algorithms are the same. The alternative hypothesis is that there is a statistically significant difference between the two sample means that are considered.

$$H_0\colon \mu_1 = \mu_2$$

$$H_A\colon \mu_i \neq \mu_2$$

Table 2.4, shows the output of the Excel analysis of the t-Test for the timing results of the In-Close3 and FCbO algorithms from *Table 2.2*.

Table 2.4, t-Test: Two-Sample Assuming Unequal Variances for the two fastest algorithms (Rank 1 and Rank 2 in Table 2.2) In-Close3 and FCbO

	CbO-PC-ICF-DBF (In-Close3)	**CbO-FC-ICF-DBF (FCbO)**
Mean	0.2530665	0.3012962
Variance	6.85845E-08	1.11922E-08
Observations	10	10
Hypothesized Mean Difference	0	
df	12	
t Stat	-539.9786481	
P(T<=t) one-tail	5.47963E-28	
t Critical one-tail	1.782287556	
P(T<=t) two-tail	1.09593E-27	
t Critical two-tail	2.17881283	

The $t_{stat}$is calculated as follows. $\bar{x}_1$and $\bar{x}_2$ are the means of sample1 and sample2 respectively. $n_1$and n_2 are the sample size for sample1 and sample2 respectively. s_1 and s_2 represent the sample deviations.

$$t_{stat} = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)}}$$

The degrees of freedom df is calculated as follows.

$$df = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{1}{n_1 - 1}\left(\frac{s_1^2}{n_1}\right) + \frac{1}{n_2 - 1}\left(\frac{s_2^2}{n_2}\right)}$$

The $P_{two-tail}$ for the student's T distribution can be found by using the T $TDistribution()$ function which requires t_{stat} and df as parameters.

$$P_{two-tail} = TDistribution(t_{stat}, df)$$

The $t_{critical\ two-tail}$ value is the inverse of the student's T probability distribution and can be obtained using the $TInverse()$ function which requires α and df as parameters.

$$t_{critical\ two-tail} = TInverse(\alpha, df)$$

Here α is the significance level.

We can reject the null hypothesis H_0 if $|t_{stat}| > t_{critical\ two-tail}$ and $P_{two-tail} < \alpha$

2.4 Detail activities undertaken in this research

After the literature review was carried out, a set of potential serial algorithms were identified. Each serial algorithm was implemented in a similar manner, taking care that any difference in performance could be only due to algorithm design and not due to implementation details or different optimizations. Next, they were tested on a 'level playing field'. Test data was carefully selected and prepared so that the algorithms could be tested on a number of key independent parameters, such as context density, number of attributes and number of objects. Details of the tests carried out are described in Section 2.5

The experiments were designed to determine the best candidate algorithm/s to take forward for parallelization. Next code profiling was used to identify parts of the program that could be optimized further. This resulted in new algorithms that can efficiently handle large datasets.

A literature review on parallel computing was carried to identify the parallel architectures that needed to be considered for implementing the parallel algorithms. A common approach used to parallelize serial algorithms is to consider hotspots in a serial implementation and to apply parallelization strategies to the serial implementation. A literature review was necessary to identify parallelization strategies adopted by other researchers for similar classes of algorithms. These strategies were distinct for different parallel architectures. Finally, the newly proposed parallel algorithms were tested with existing algorithms using an experimental approach similar to what was used earlier.

2.5 Benchmarking of the implementations

Other research have reiterated the importance of having a set of publically available standard datasets for comparing FCA algorithms (Andrews, 2009; Kuznetsov & Obiedkov, 2002). Although there are no standard dataset established for testing FCA algorithms several research studies have used publicly available datasets for this purpose (Andrews, 2009, 2011, 2015; Frank & Asuncion, 2010; Krajca, Outrata, & Vychodil, 2008; Krajca et al., 2010; Strok & Neznanov, 2010). In this research several popular public datasets such as the mushroom dataset, adult dataset from the UCI learning repository were used.

Table 2.5, shows how the performance of different algorithms for benchmarked datasets used in FCA literature would be depicted. In addition, random data experiments were also carried out.

Table 2.5, Sample test cases for actual datasets (timing in seconds)

Data Set	**Mushroom (UCI)**	**Adult (UCI)**	**Internet Ads (UCI)**	**Student (SHU)**
|G| x |M|	8,124x125	32,561x99	3279 x 1565	587 x 145
Density	17.4%	11.29%	0.97%	24.50%
# Concepts	226,920	80,332	16,570	22,760,242
Algorithm 1	t_{11}	t_{21}	t_{31}	t_{n1}
Algorithm 2	t_{12}	t_{22}	t_{32}	t_{n2}
…				
Algorithm m	t_{1m}	t_{2m}	t_{3m}	t_{nm}

A random dataset generator used in the FCA community was used to generate random datasets of particular characteristics. The generator can be used to generate a random dataset that matches the given number of objects, number of attributes and density. The generated dataset has a filename that indicates its characteristics. For instance, the random dataset n100m20000d5s1000.cxt has 100 attributes (n), 20,000 objects (m) and has a density of 5%. The number 1000 after the latter 's' indicates the random generator seed used. The same generated dataset file was used in all experiments that required a dataset of 100 attributes, 20,000 objects with a density of 5%. The number of concepts that is there for a given dataset is dependent on the number of attributes, number of objects and the density. It is also dependent on the arrangement of crosses that are there in the dataset that represents the binary incidence relationship between the number of objects and the number of attributes. In general, given a dataset of |G| number of objects the number of concepts at most would be $2^{|G|}$.

The following large random datasets used in the FCA community was also used as benchmarking datasets. The M7X10G120K, M10X30G120K and the T10I4D100K datasets have the following characteristics (See *Table 2.6*). Here |G| and |M| represent the number of objects and the number of attributes respectively.

Table 2.6, Random Benchmarking datasets

Data Set	M710G120K	M10X30G120K	T1014D100K
|G| x |M|	120,000 x 70	120,000 x 300	100,000 x1,000
Density	10.00%	03.33%	01.01%
# Concepts	1,166,343	4,570,498	2,347,376

2.6 Unit Testing the Implementations

All the implementations developed had a common core handling reading the dataset and generating an output file containing all the generated concepts.

The following datasets were used in order to unit test the implementations. Sample Dataset (Andrews, 2015), Tealady dataset, Water Lilies dataset (Ganter & Wille, 1999), n100m100d5s1000.cxt, n200m100d5s1000.cxt and the mushroom. These datasets have 10, 65, 112, 395, 1,062, 226,921 concepts respectively. The output file generated from the implementations were compared using the diff utility with the output file generated by an open source implementation of InClose[1] for each of the datasets. If there was any deviation noted in diff it was assumed that the unit test had failed. In a majority of cases the difficulty was to get the implementation to pass the simpler datasets upto Water Lilies. Later once the artificial dataset unit tests were passed, the implementations usually passed the mushroom real-world dataset without any modification. This approach was used to test all the serial, shared memory parallel and the distributed parallel algorithm implementations.

2.7 Comparison of Serial Implementations of FCA

Here eight variants of Kuznetov's CbO algorithms were compared. The eight algorithms considered were implemented in a level playing field. Five of the algorithms were based on the re-specification of the algorithms by Andrews (2014). The remaining three new algorithms were also represented using the same notation used by Andrews. The importance of comparing algorithms which are represented in the same level of abstraction was highlighted by

Kuznetsov and Obedkov (2001). Andrews (2014) re-specification of the algorithms are described in the same level of abstraction. Each of the different code blocks used in the eight CbO algorithms were implemented as C++ functions. The eight algorithms had 17 unique code blocks which were implemented as 16 separate C++ functions. (Two code blocks mapped into one C++ function.) The algorithms were finally implemented in C++ by assembling the C++ functions developed. The algorithms were implemented in an un-optimized way to focus on the comparison of the algorithms themselves and not on any efficiencies provided by optimizing the code. All implementations used the same data structures for representing objects, attributes and the context. The details of the different data structures used are given in Table 4.2, Kuznetsov and Obedkov (2001) also raised the importance of using similar data structures when comparing different algorithms.

The eight algorithms were implemented in an un-optimized way. An un-optimized implementation of an algorithm is also a very good starting point for subsequent optimization and parallelization. Knuth (1974) famously said that premature optimization is the root of all evil. Jackson (1975) has also cautioned against early optimization which can result in a design that is not as clean as it could have been or code that is not correct, as a result of the complicated code due to the optimizations carried out.

The approach used to implement the eight algorithms using common code blocks ensured that the programs correlated to the algorithms in a similar way. Empirical testing was carried out on the code compiled with the Linux Intel C++ Compiler version 15. The implemented algorithms were tested on Intel ® AI Cloud[2] running in a Colfax Cluster[3] on a Compute Node that used an Intel® Xeon ® Gold 6128 @ 3.7 GHz, six core processor with 96GB RAM, running a stripped down version of Suse Linux. Benchmarking of the programs were carried out by making use of three real world datasets (Mushroom, Adult, Internet Ad) which have been used widely by the FCA community and a series of artificial datasets. The artificial datasets were collected to highlight the variation of the independent parameters density, the number of objectes and the number of attributes. To see the effect of the variation of density across the five programs, a series of artificial datasets were used where the density varied from 25 to 50. In each of these datasets the number of attributes and the number of objects were fixed to 100 (See *Table 4.5).* The variation of objects were examined by a series of artificial datasets where the number of objects varied from 10,000 to 100,000. Here the number of attributes and the density were fixed at 100 and 5% respectively (See *Table 4.6).* Finally, the variation of attributes was observed by a series of artificial datasets where the number of attributes varied from 1000 to 2000. The number of objects and the density were fixed at 100 and 5% respectively (See *Table 4.7*), All the programs used in comparing the serial implementations were compiled in debug mode with no optimization (compiled using the Intel Compiler flags `-O0 -g`) to ensure that the compiled code represented a true reflection of the high level algorithms that were compared. To verify that the computer and compiler used for the experiments had no side effect in the results the three real world datasets were tested again on three other computers (an Intel Core i5-5210M CPU @ 2.5GHz, with 8GB RAM, running a Microsoft Windows 7 - 64 bit operating system and a Core i7-4650U CPU @ 3.3GHz with 8GB RAM running Microsoft Windows 8 – 64 bit operating system) with two windows compilers (Microsoft Visual Studio 2010 and Intel C++ Compiler version 16). Each machine ran each of the programs compiled by both the Intel and the Microsoft compilers. The experiments were also executed on the Archer Super Computer which has a node consisting of two two 2.7 GHz, 12-core Intel ® Xeon ® E5-2697 v2 processors with 64GB RAM each. A Cray compiler was used for the compilation. The results obtained had the same variation as the results shown in Section 4.4.

2.8 Experimental Testing of Parallel Algorithms

The parallel algorithms that are proposed were experimentally validated and tested using a similar approach to the Serial programs. The scalability of parallel programs is represented by the following formulae.

$$S_p = \frac{T_1}{T_p}$$

Where T_1 is the time taken to execute a program in one processor and T_p the time taken to execute a program on p number of processors. A graph can be used to show how scalable the implementation is.

Here too all samples were statistically analysed as mentioned in Section 2.2.

The OpenMP shared memory implementations were tested using the Intel ® AI Cloud mentioned in Section 2.7

The MPI implementation of the Parallel In-Close3 algorithm was executed on 12 nodes of the ARCHER Super Computer. An ARCHER node has two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series Xeon processors. The two processors are connected by two QuickPath Interconnect (QPI) links. The memory is arranged in a non-uniform access (NUMA) form, where each 12-core processor has a single NUMA region with local memory of 32 GB. The Cray Aries interconnect links all compute nodes in a Dragonfly topology. In the Dragonfly topology four compute nodes are connected to each Aries router; 188 nodes are grouped into a cabinet; and two cabinets make up a group. (Turner & McIntosh-Smith, 2017). By nature a super computer provides dedicated access of the compute nodes required to run a program. This ensures that during the empirical testing that all the test results reflects only the execution time of the programs that were run.

[1] https://sourceforge.net/projects/inclose/
[2] https://ai.intel.com/devcloud/
[3] https://colfaxresearch.com

3. FORMAL CONCEPT ANALYSIS

3.1 Formal Concept Analysis Background

Knowledge Representation is the field in artificial intelligence which focuses on the representation of information, that can be used by a computer to solve complex problems. Ontologies, which capture the relationships between concepts is a knowledge representation technique. Ontologies are generally created by domain experts (Uschold & Gruninger, 1996). There are significant advantages if ontologies can be created automatically from structured data (Alani et al., 2003). A form of clustering algorithms called biclustering algorithms can be essentially used for this purpose. Clustering is the task of grouping related sets of data together (Kaytoue, Kuznetsov, & Napoli, 2011). Even in situations where the clustering algorithms are only able to capture simpler relationships such as an is-a relationship, this still has significant practical applications in many domains.

Formal Concept Analysis can be thought of as a form of biclustering which can generate is-a relationships from structured binary tabular data. The is-a relationship forms the basis of grouping concepts into super concepts and sub concepts and is represented as a concept lattice. Implications and Associate Rules are direct by products of a concept lattice (Wille, 2005b).

The origins of concept hierarchies go way back to Aristotle (384 BC – 322 BC) who introduced the important concept of taxonomies in his books Metaphysics and Categories. His objects were hierarchically organised according to properties (Cimiano, 2009). The concepts defined in the “classical theory of concepts” are mathematically represented as formal concepts in formal concept analysis (FCA). In psychology/philosophy concepts are formally definable by its features. This approach is still a popular way to represent concepts although the “classical theory of concepts” doesn’t accurately represent human cognition. The concepts found in FCA are called “formal concepts” to avoid confusion non-mathematical definition of concepts(Priss, 2006).

Knowledge is represented in formal concept analysis as a formal context. This describes a binary relationship between a set of objects and a set of attributes of a domain.

A precise definition of a formal context is given below.

A formal context is defined as $K := (G, M, I)$

Where G is a set of objects, M a set of attributes and I a binary incidence relationship between G and M with $I \subseteq G \times M$.

Since formal contexts are a binary relationship, they can be represented as cross tables. Here each object and attribute is represented as a row and a column respectively (Ganter, Stumme, & Wille, 2002; Wille, 1982).

Let’s take an example borrowed from Krötzsch to illustrate formal concept definitions making use of the planets in our Solar System (Krötzsch & Ganter, 2009). *Figure 3.1* shows the Formal Context of eight major planets in our Solar System with details of seven different properties. The table has the eight planet as rows and the seven properties as columns. A cross is used to mark the presence of a property against a planet. For example, the planets Jupiter, Saturn, Uranus, and Neptune are considered gas giant planets. This is denoted in the table with crosses marked against each of the planets mentioned earlier for the property gas giant planet. The rows are known as objects and the columns are known as attributes. A formal concept is defined as a pair of maximum of a set of objects and a set of attributes. For the earlier example

	Has some moon	Gas giant	Sun sets in the west	Average magnetic Field weaker than earth	Was Known as a planet in Ancient Rome	Average Temperature above 0°C	Stable atmosphere contains Nitrogen
	a	b	c	d	e	f	g
1. Mercury			X	X	X	X	
2. Venus				X	X	X	X
3. Earth	X		X		X	X	X
4. Mars	X		X	X	X		X
5. Jupiter	X	X	X		X		
6. Saturn	X	X	X	X	X		
7. Uranus	X	X					
8. Neptune	X	X	X				

Figure 3.1, A Formal context about planets in our solar system (Krötzsch & Bernhard, 2009)

we can also see that the four planets also have the property *Has Some Moon* checked. So the formal concept consisting of gas giants is defined as *({Jupiter, Saturn, Uranus, Neptune}, {Gas Giant, Has Some Moon})*. Similarly, we can define a formal concept of planets that has some moon. From *Figure 3.2* we can find the following concept *({Earth, Mars, Jupiter, Saturn, Uranus, Neptune}, {Has some Moon})*. Unlike the previous example, there are no additional common properties for the planets considered. The planets Earth, Mars, Jupiter, Saturn, Uranus, and Neptune are the planets that has some moon. The planets mentioned are the extent of this formal concept and the attribute form the intent. A mathematical definition of Formal Concepts is given below.

For a set of objects $A \subseteq G$ the set A' is defined as

$$A' := \{m \in M \mid m\ I\ g \text{ for all } g \in A\}$$

Similarly for a set of attributes $B \subseteq G$ the set B' is defined as

$$B' := \{g \in G \mid m\ I\ g \text{ for all } m \in B\}$$

(A, B) is a formal concept if $A' = B$ and $B' = A$.

There are many formal concepts in a formal context. All the possible formal concepts that are there in a formal context can be generated and be represented in a concept hierarchy. A concept lattice can be used to represent a concept hierarchy (See *Figure 3.2*).

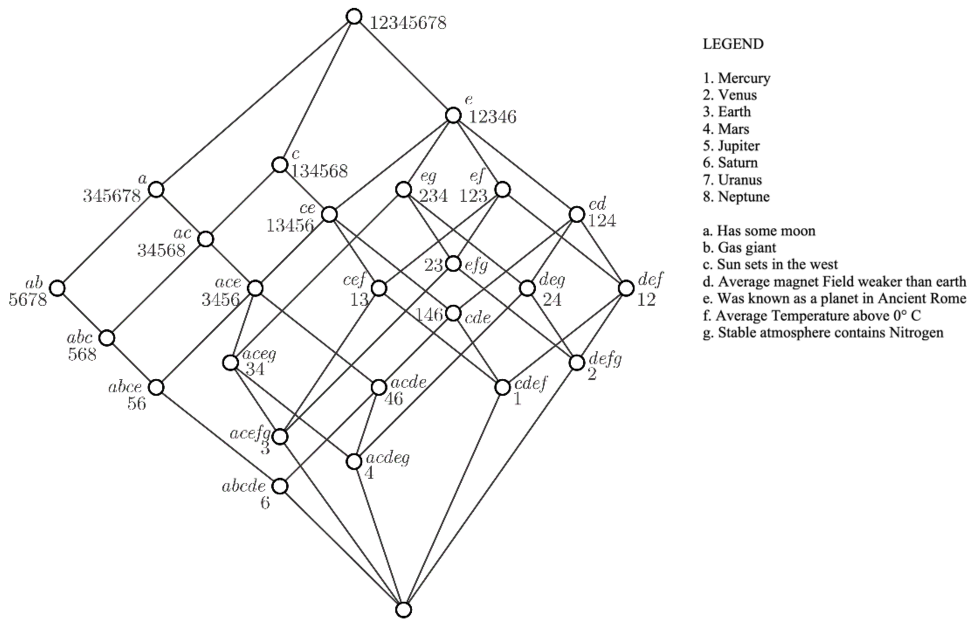


Figure 3.2, The Concept Lattice for the Formal Context given in Fig 3.1 (Krötzsch & Bernhard, 2009)

Here the object and attribute indices *1* to *8* and *a* to *g* are used for brevity. The two examples mentioned earlier map to the concepts *({5,6,7,8},{a,b})* and *({3,4,5,6,7,8},{a})* respectively in the Lattice. They are the left most vertex of the lattice and the vertex immediately above. In the example taken the concept *({Earth, Mars, Jupiter, Saturn, Uranus, Neptune}, {Has some Moon}) (Shown as ({3,4,5,6,7,8},{a}) in lattice)* is the sub concept of *({Jupiter, Saturn, Uranus, Neptune}, {Gas Giant, Has Some Moon}) (Shown as ({5,6,7,8},{a,b} in lattice).* Accordingly in the concept lattice the concept *({3,4,5,6,7,8},{a})* is above the concept *({5,6,7,8},{a,b})* as it is more general. A mathematical definition of the relationships between formal concepts is given below.

Let $K := (G, M, I)$ define a formal context. Assuming (A_1, B_1) and (A_2, B_2) are formal concepts of K.

$$(A_1, B_1) \leq (A_2, B_2) := A_1 \subseteq A_2 (\Leftrightarrow B_1 \supseteq B_2)$$

(A_1, B_1) is called the sub concept of (A_2, B_2), and (A_2, B_2) is called the super concept of (A_1, B_1). In the Concept Lattice the more general concepts are above and the specific concepts are below. The set of all formal concepts of K together with the defined order relation is denoted by $\beta(G, M, I)$, which can be represented as a Concept Lattice (See *Figure 3.2*)

FCA has evolved into an unsupervised learning method for discovering conceptual structures in data. Conceptual hierarchies allow the analysis of complex structures and the discovery of dependencies within the data (Sarmah, Hazarika, & Sinha, 2015). Unsupervised learning is a sub branch of artificial intelligence. FCA has been applied a wide range of disciplines. Wille who developed FCA in the 1980s presents details of the initial applications of FCA by members of his TU Darmstadt research group. These mainly were applications that support knowledge communication (Wille, 2005a). A comprehensive survey of the usage of FCA in the area of Knowledge Processing in a wide range of domains which includes software mining, web analytics, medicine, biology and chemistry data are presented in (Poelmans, Ignatov, Kuznetsov, & Dedene, 2013b, 2013a). Carpineto and Rodmano outline the use of FCA in applications that use information retrieval, text mining and rule mining (Carpineto & Romano, 2004). FCA has also been used extensively in Software Engineering. These include applications in Requirement Analysis, Architectural Design, Detail Design and Software Maintenance (Tilley, Cole, Becker, & Eklund, 2005). FCA has been used to analyse the body of knowledge in Software Architecture by analysing research papers in that domain (Couto, Oliveira, Ferreira, & Bouwers, 2011). Use of FCA in interactive user applications are described in the development of interactive museum applications for searching virtual objects and physically navigating museum exhibits by Wray and Eklund (Wray & Eklund, 2011, 2014).

3.2 FCA Algorithms

There are several well-known algorithms that generate a set of all the formal concepts that are there in a formal context. The number of concepts is known to be exponential of the size of the input context (Kuznetsov & Obiedkov, 2002). Algorithms that generate formal concepts broadly fall into two categories

(a) Generate only the list of concepts
(b) Generate a concept lattice.

Kuznetsov compares the efficiency of ten algorithms that generate formal concepts (Kuznetsov & Obiedkov, 2002). Some of the algorithms compared included NextClosure (Ganter, 1984), Bordat (Bordat, 1986), Close by One (Kuznetsov & Obiedkov, 2002), Lindig (Lindig, 2000), Chein (Chein, 1969), Nourine (Nourine & Raynaud, 1999), Godin (Godin, Missaoui, & Alaoui, 1995), Dowling (Dowling, 1993) and Titanic (Stumme, Taouil, Bastide, Pasquier, & Lakhal, 2000). Different algorithms use different strategies to generate a new intent. Some algorithms compute an intent explicitly by intersecting all the objects of the corresponding extent. Others intersect a generated intent with some object's intent (Kuznetsov & Obiedkov, 2002).

A canonicity test is used by FCA algorithms to check if the current concept is being generated for the first time. In some of the algorithms complete closure of a concept is needed before the canonicity test, and in some the testing of the canonicity can take place before the complete closure (partial closure). Certain algorithms also keep track of the canonicity test failures which can be applied before closure (Andrews, 2015; Outrata & Vychodil, 2012).

The algorithms presented in this thesis focus only on generating all the concepts of a given context. It is sufficient to generate all the concepts of a given context instead of the generation of concept lattices in certain situations. These include applications where visual representation of concepts hierarchies are not required, to situations where alternative graphical representations can be used. An example of an alternative graphical representation to a lattice is concept trees (Andrews & Hirsch, 2016). In any case concept lattices are effective as a visual representation of concept hierarchies for representing only a small number of concepts. Computing formal concepts is a fundamental study in its own right and there are a number of algorithms that focus on this without computing the lattice (Outrata & Vychodil, 2012; Strok & Neznanov, 2010).

3.3 Limitations of Serial FCA Algorithms

Old and Priss have highlighted the need of developing algorithms that can handle large contexts (Old & Priss, 2004). There is no formal definition of what a large context is, however based on recent benchmarks a large context can be taken as consisting of over 100,000 objects and over 100 attributes resulting in millions of concepts (Andrews, 2017; Outrata, 2015). In real world datasets the size of attributes are limited in size. There is an explosion of data that is produced daily. In 2012, a study carried out by IBM estimated that 2.5 exabytes of data are produced daily (Wall, 2014). Structured data is still a significant type of data that is used in organizations (Desiere, 2015). FCA has been extended into areas such as data mining, web mining which involves large data sets (Fu & Nguifo, 2004). Datasets in domains such as genomics, internet of things, log analysis are significantly large and are growing at a rapid pace. For instance Genomics dataset sizes have been doubling every 18 months (Langmead & Nellore, 2018). There is a necessity for FCA algorithms to handle this volume of data. The possession of a fast algorithm for computing or updating the underlying concept lattice is an essential prerequisite in many applications for instance rule discovery, document ranking and program analysis (de Moraes, Dias, Freitas, & Zárate, 2016; Poelmans et al., 2013a, 2013b).

All computing devices used today are parallel machines. The introduction of multicore processors commenced around the year 2004 to solve the so-called power wall problem (See *Figure 3.3*). Prior to this CPU manufacturers resorted to increase the clock speed of each new generation of CPU eventually reaching the critical power consumption of 130 Watts around 2004. Beyond this point, it was not economically possible to dissipate the heat produced by the CPU's. Over the last decade CPU manufacturers have kept the clock speed and core size of a CPU as constants and have resorted instead to add extra cores to a single die in the CPU to get better performance (Chappell & Stokes, 2012).

Today's laptops, desktop machines have at least two to four cores in the CPU. High end Xeon Processors have up to 24 cores (Jeffers et al., 2016). Computer programs must be designed and implemented using a parallel approach to leverage on the multiple cores available in the CPU (Sutter, 2005). Traditional serial programs can make use of only one CPU core of the computer. In essence, an algorithm that needs its implementation to fully utilize the hardware of a modern computer needs to be a parallel algorithm. With the rapid increase of datasets that require analysis, the next generation FCA algorithms should be parallel in nature.

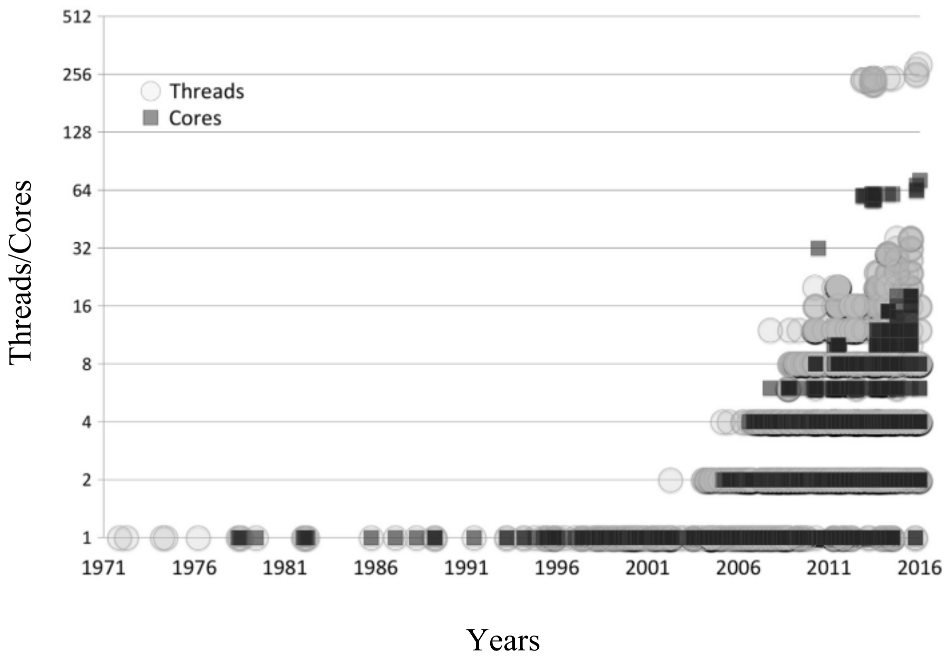


Figure 3.3, Processor/coprocessor core/thread parallelism (log scale) (Jeffers, Reinders, & Sodani, 2016)

3.4 Existing Parallel FCA Algorithms

Huaiguo Fu had created a parallel implementation of the NextClosure algorithm but it was limited to 50 attributes (Fu & Nguifo, 2004) this was subsequently greatly extended (Fu & Foghlu, 2008). Krajca (Krajca, Outrata, & Vychodil, 2008) presented a parallel algorithm called PFCbO which parallelizes the FCbO algorithm. This is also a variation of the CbO algorithm (Kuznetsov & Obiedkov, 2002). Krajca has also presented a distributed version of this algorithm that uses the Map Reduce distributed framework (Krajca & Vychodil, 2009).

The main contribution of the thesis are the new parallel FCA algorithms for shared memory and distributed memory models which are presented in Chapter 5 and Chapter 6 respectively.

4. COMPARISION OF FCA SERIAL ALGORITHMS

4.1 Existing FCA Algorithms

Many of the classical algorithms such as Bordat, NextClosure, Chein, Lindig and Nourine are batch algorithms. These algorithms generate the entire context from scratch. Some of the other algorithms including Godin, Downling, Norris, Close by One (CbO), Kracja and In-Close are incremental algorithms. Incremental algorithms produces the concept set for the first ith concepts at the ith step.(Chang-Sheng, Jing, Hai-Long, Long-chang, & Bing-ru, 2013; Gajdoš & Snášel, 2014; Sarmah et al., 2015).

The generation of formal concepts involves looking at how to generate all the formal concepts and avoiding the generation of the same concept. FCA algorithms such as Chein, Norris (Norris, 1978), Lindig and Godin avoided the regeneration of concepts by searching through generated concepts (Berry, Bordat, & Sigayret, 2007). Algorithms such as NextClosure, Close by One (CbO) makes use of a lexicographical order in the generation of concepts and thus avoids searching through the generated concepts (Berry et al., 2007). Out of the many algorithms that have been developed (Andrews, 2009, 2011, 2015, Krajca, Outrata, & Vychodil, 2008, 2010; Outrata & Vychodil, 2012; Strok & Neznanov, 2010), have noted that the CbO based algorithms produce the best results for different types of datasets.

Algorithms are described by their authors at different levels of abstraction. Some algorithms such as CbO are described by Kuznetsov using an abstract mathematical set notation (Kuznetsov & Obiedkov, 2002), while others such as FCbO (Outrata & Vychodil, 2012) or In-close (Andrews, 2009) are described in pseudo code which is closer to the implementation level in a programming language. Algorithms which are described in a higher abstract notation would be interpreted subjectively when other researchers implement them. A proper comparison of a set of algorithms can be carried out if they are described at the same level of abstraction. Andrews (2014) presents the five variations of the CbO algorithms using the same level of abstraction. In order to highlight the effects of the key features used in recent variations of the CbO algorithm this thesis introduces three new variations of the CbO algorithm presented using the same notation used by Andrews. They are the CbO full closure using a combined depth first and breadth first search (CbO-FC-BDF), CbO full closure using inherited canonicity test failures and depth first search (CbO-FC-ICF-DF), and CbO partial closure with incremental closing of intents using inherited canonicity test failures and depth first search (CbO-PC-ICF-DF).

This chapter presents the detail comparison of the three main enhancements that have been applied to Kusnotsov's CbO algorithm. The three main enhancements considered are the partial closure with incremental closure of intents, inherited canonicity test failures and using a combined depth first and breadth first search. Eight variations of the CbO algorithm are presented and implemented in an un-optimised way allowing the identification of the best algorithms to select for future algorithmic enhancement such as parallelization.

Krajca, Outrata, and Vychodil (2010); Andrews (2014); Outrata and Vychodil (2012) have recently compared optimized implementations of the CbO family of FCA algorithms. The optimized implementations were developed by their respective authors. Because the implementation and the optimization of the algorithms can be done in many ways, the implementations may not be a true representations of the high level algorithms described in pseudo code to carry out a scientific comparison of the algorithms. Kuznetsov & Obiedkov (2002) compared existing FCA algorithms in 2002 making use of a similar approach used in this thesis. Five of the algorithms compared in this thesis are contemporary algorithms which have been developed during the last six years (Since 2009). The older algorithms do not fare well with the contemporary ones (Outrata & Vychodil, 2012; Strok & Neznanov, 2010)

Table 4.1 highlight the summary of the eight CbO variants presented in this paper. CbO-FC-DF is the base algorithm used, the rest are carefully selected variants that highlight the three key enhancements used.

Table 4.1, Eight Variations of CbO

Algorithm	**Originally Described in**	**Use of a Queue**	**Partial Closures**	**Inheritance of failed canonicity tests.**
CbO-FC-DF	Krajca (Vychodil, 2008)			
CbO-PC-DF	In-Close (Andrews, 2009)		X	
CbO-FC-DBF	New	X		
CbO-PC-DBF	In-Close2 (Andrews, 2011)	X	X	
CbO-FC-ICF-DF	New			X
CbO-PC-ICF-DF	New		X	X
CbO-FC-ICF-DBF	FCbO (Krajca, Outrata, & Vychodil, 2010)	X		X
CbO-PC-ICF-DBF	In-Close3 (Andrews, 2014)	X	X	X

The algorithms have been named making use of the following abbreviations FC – full closure, PC – partial closure with incremental closure of attributes, DF – depth first search, DBF – combined depth and breadth search and ICF – inherited canonicity test failure. Algorithms that use the combined depth and breadth search feature (DBF) make use of a queue data structure.

The CbO-PC-DF, CbO-FC-DBF and the CbO-FC-ICF-DF algorithms make use of the three enhancements, partial closure with the incremental closure of attributes (PC), combined depth and breadth search (DBF) and the inherited canonicity test failure (ICF) respectively in isolation. The CbO-PC-DBF, CbO-PC-ICF-DF and CbO-FC-ICF-DBF algorithms make use combining two of the three enhancements as shown in *Table 4.1.* Finally the algorithm CbO-PC-ICF-DBF combines all the three enhancements. The CbO-FC-DBF, CbO-PC-ICF-DF and CbO-FC-ICF-DF are new algorithms presented in this thesis. The remaining five algorithms are existing algorithms and details of their original sources are given in *Table 4.1.*

4.2 Details of the Eight CbO based Algorithms

4.2.1 CbO-FC-DF (CbO – Full Closure, Depth First Search Algorithm)

```
CbO-FC-DF (CbO)
ComputeConceptsFrom($(A, B), y$)
1 ProcessConcept($(A, B)$)
2 for $j \leftarrow y$ upto $n - 1$ do
3     if $j \notin B$ then
4         $C \leftarrow A \cap \{j\}^{\downarrow}$
5         $D \leftarrow C^{\uparrow}$
6         if $B \cap Y_j = D \cap Y_j$ then
7             ComputeConceptsFrom($(C, D), j + 1$)
```

Figure 4.1, CbO-FC-DF (CbO) algorithm pseudo code (Vychodil, 2008)

The CbO-FC-DF (CbO – Full Closure, Depth First Search) algorithm shown in *Figure 4.1*, was originally described by Vychodil (2008). In our discussion we will use this as our base algorithms. Here (A, B) is the concept generated where A is the extent and B is the intent. n is the number of attributes in the context and y is the attribute that is currently being considered. The algorithm is invoked with $(A, B) = (X, X^{\uparrow})$. Where X represents a complete set of extents. Recalculation of already computed concepts can be avoided by the statements in line 3 and line 6.

$$j \notin B$$

$j \notin B$ enables skipping attributes in the current intent (Vychodil, 2008). Here if the currently considered attribute j is already a member of the currently considered intent then the extent of this has already been computed. This is due to the following observation.

$$\{0,1,2,,,k,,n\}^{\downarrow} = \{0,1,2,,,k,,n\}^{\downarrow} \cap \{k\}^{\downarrow}$$

The canonicity test is defined by the condition given in line 6.

$$B \cap Y_j = D \cap Y_j$$

Here Y_j is a set containing all the attributes upto attribute j. Due to the lexical order of computing the concepts, the above condition becomes false only if D is lexically before B. Thus it implies that the concept has been computed before and can be skipped. The extents and the intents are computed in line 4 and 5 respectively. A full closure is used in calculating the intent in line 5.

$$D \leftarrow C^{\uparrow}$$

The complexity of CbO family of algorithms are $O(|G|^2.|M|)$ (Kuznetsov & Obiedkov, 2002). The implementation of CbO is given in *Figure 4.2* to illustrate how the common code blocks are used in the actual algorithm implementation. Lines 2 to 8 of the implementation of CbO-FC-DF contain declaration of variables. Lines 10 to 19 directly correspond to the CbO-FC-DF algorithm given in *Figure 4.1*. *Table 4.2*, shows details of the 17 code blocks used in the algorithms and their corresponding C++ functions used in the implementation. The `VECTOR_TYPE` used in the code is a macro representing an unsigned 64 bit integer which is used to store binary values.

Table 4.2, List of Functions used to implement the common code blocks

Code Blocks	Common Function used in Implementation of the five algorithms
$j \notin B$	`bool isMember(int j, VECTOR_TYPE *BBit)`
$C \leftarrow A \cup \{j\}^{\downarrow}$	`void aIntersectionColj(int **C,int &CSize,int *A ,int ASize,int j)`
$D \leftarrow C^{\uparrow}$	`void deriveIntentsBit(short int **B, int &BSize, VECTOR_TYPE *BBit, int *A, int ASize)`
$B \cap Y_j = D \cap Y_j$	`bool isBequalToDuptojBit(VECTOR_TYPE *BBit, VECTOR_TYPE *DBit, int j)`
$A = C$	`bool isEqual(int I1[],int I1Size,int I2[], int I2Size)`
$B \leftarrow B \cup \{j\}$	`void insert(VECTOR_TYPE *BBit, short int B[], int &BSize, int j)`
$B = C^{\uparrow Y_j}$	`bool buptoJisEqualtoPartialClosureOfCuptoJBit(VECTOR_TYPE *BBit, int C[], int CSize, int j)int C[], int CSize, int j)`
$B \cap Y_j = C^{\uparrow Y_j}$	`bool buptoJisEqualtoPartialClosureOfCuptoJBit2(VECTOR_TYPE *BBit, int C[], int CSize, int j)`
$PutInQueue(C,j)$	`void put(TQueue &q, int *Av, int ASize,  int BSize, int j)`
$GetFromQueue(C,j)$	`bool get(TQueue &q, int Av[], int &ASize, int &BSize, int &j)`
$M^j \leftarrow N^j$	`void copyRowArrMN(VECTOR_TYPE **M, VECTOR_TYPE *D)`

$N^j \cap Y_j \subset B \cap Y_j$	`bool isNjSubSetofBuptoJBit(VECTOR_TYPE *NBit, VECTOR_TYPE *BBit, int j)`
$M^j \leftarrow D$	`void copyRowArrMN(VECTOR_TYPE **M, VECTOR_TYPE *D)`
$PutInQueue((C,D),j)$	`void put(TQueue &q, int *Av, int ASize, short int *Bv, int BSize, VECTOR_TYPE *BBitv, int j)`
$GetFromQueue((C,D),j)$	`bool get(TQueue &q, int **Av, int &ASize, short int **Bv, int &BSize, VECTOR_TYPE **BBitv, int &j, short int &level)`
$M^j \leftarrow C^{\uparrow Y_j}$	`void copyRowArrMPartialClosure(VECTOR_TYPE **M,  int j, int A[], int ASize)`
$D \leftarrow B \cup \{j\}$	`void copyInsert(VECTOR_TYPE *BBit, short int D[],int &DSize, short int B[], int BSize, int j)`

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize, VECTOR_TYPE *BBit,
                         int y) {
    int C[OBJECTSIZE];
    short int D[ATTRIBUTESIZE];
    VECTOR_TYPE DBit[VECTOR_MAX_COLS_CELLS];
    int CSize, DSize;
    // Print Concept
    conceptno++;
    int concept = conceptno;

    for (int j=y; j<n; j++) {
       if (! isMember(j, BBit)) {
           aIntersectionColj(C,CSize,A,ASize,j);
           deriveIntentsBit(D,DSize, DBit, C, CSize);
           if (isBequalToDuptojBit(BBit, DBit, j)) {
              ComputeConceptsFrom(C, CSize, D, DSize, DBit, j+1);
           }
       }
    }
}
```

Figure 4.2, Listing of CbO-FC-DF program code

We will next apply the three enhancements in isolation to the base algorithm CbO-FC-DF. First we will apply the partial closure with incremental closure of intents to the base algorithm. The resulting algorithm CbO-PC-DF (CbO – Partial Closure, Depth First Search) algorithm was originally described by Andrews (2009) as the In-Close algorithm (See *Figure 4.3)* The key difference in CbO-PC-DF compared to CbO-FC-DF are the incremental closures given in line 4, 7 (See *Figure 4.4)* and the new partial closure canonicity test given in line 6. This is tested before the concept is closed.

$$B = C^{\uparrow Y_j}$$

Here the context I is examined upto the current attribute j. A full closure operator $\uparrow$ is equivalent to $\uparrow Y$ where Y is the set of all attributes in context I. The partial closure operator $\uparrow Y$ is defined as follows (Andrews, 2014).

$$A^{\uparrow Z} := \{y \in Z \mid \forall x \in A : x\, I\, y\}$$

4.2.2 CbO-PC-DF (CbO – Partial Closure, Depth First Search Algorithm)

CbO-PC-DF (In-Close)

```
ComputeConceptsFrom($(A, B), y$)
1 for $j \leftarrow y$ upto $n - 1$ do
2     $C \leftarrow A \cap \{j\}^{\downarrow}$
3     if $A = C$ then
4         $B \leftarrow B \cup \{j\}$
5     else
6         if $B = C^{\uparrow Y_j}$ then
7             $D \leftarrow B \cup \{j\}$
8             ComputeConceptsFrom($(C, D), j + 1$)
9 ProcessConcept($(A, B)$)
```

Figure 4.3, CbO-PC-DF (In-Close) algorithm pseudo code (Andrews, 2009)

Due to the incremental closure of intents, it's not possible to skip attributes as in the algorithm CbO. Hence $j \notin B$ which appears in Line 3 of the CbO-FC-DF (See *Figure 4.1*) cannot be used in the CbO-PC-DF algorithm. The implementation of the CbO-PC-DF algorithm is given in *Figure 4.4*. Lines 9 to 23 correspond directly to the CbO-PC-DF algorithm given in *Figure 4.3*

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize,
                              VECTOR_TYPE *BParentBit, int y) {
   int C[OBJECTSIZE];
   short int D[ATTRIBUTESIZE];
   int CSize, BSize;
   VECTOR_TYPE BChildBit[VECTOR_MAX_COLS_CELLS]; //the current intent in Boolean form
   memcpy(BChildBit,BParentBit,nArray* VECTOR_SIZE_BYTES);
   int concept = conceptno; // keeps track of the current concept

   for (int j=y; j<=n; j++) {
      aIntersectionColj(C,CSize,A,ASize,j);
      if (isEqual(A, ASize, C,  CSize)) {
         insert(BChildBit,B,BSize,j);
      }
      else {
         if (buptoJisEqualtoPartialClosureOfCuptoJBit(BChildBit,C,CSize,j)) {
            copyInsert(BChildBit, D, DSize, B, BSize, j)
            ComputeConceptsFrom(C,CSize, D, DSize, BChildBit, j+1);
          }
      }
   }
   // Print Concept
   conceptno++;
}
```

Figure 4.4, Listing of CbO-PC-DF program code

Both CbO-FC-DF and the CbO-PC-DF algorithms use a depth first search stratergy by invoking the recursion call to compute the next concept as soon as details of the next intent to be considered is found. *Figure 4.5(a)* shows a depth first recursive call tree. The number represents the order in which the recursive function is called. The second enhancement is to consider the effects of delaying the recursive call by using a combined depth first and breadth first search stratergy. *Figure 4.5(b)* shows how a combined depth first and breadth first recursive stratergy works. This can be implemented by storing the intents to be considered in a queue and calling the recursive function only once all the intents to be consided in a given level of recursion are computed.

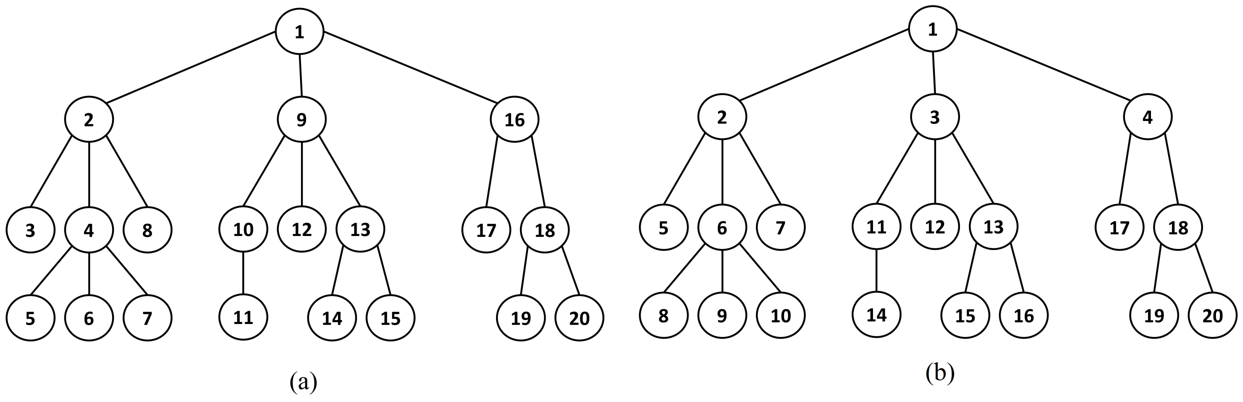


Figure 4.5, (a) Depth First Recursive Tree and Depth and (b) Breadth First Recursive Tree

The next algorithm presented CbO-FC-DBF (CbO – Full Closure, Combined Depth First and Breadth First Search) (See *Figure 4.6*) extends the CbO-FC-DF algorithm by incorporating a combined depth first and breadth first searching stratergy by incorporating a queue. The `PutInQueue()` function in line 7 of the algorithm stores the next intent to be considered in a queue. The recursive call to compute the next concept is carried out later in line 9 of the algorithm.

4.2.3 CbO-FC-DBF (CbO – Full Closure, Combined Depth First and Breadth First Search Algorithm)

```
CbO-FC-DBF
ComputeConceptsFrom(($A, B$), $y$)

1 ProcessConcept(($A, B$))
2 for $j \leftarrow y$ upto $n - 1$ do
3     if $j \notin B$ then
4         $C \leftarrow A \cap \{j\}^{\downarrow}$
5         $D \leftarrow C^{\uparrow}$
6         if $B \cap Y_j = D \cap Y_j$ then
7             PutInQueue (($C, D$), $j$)
8 while GetFromQueue(($C, D$), $j$) do
9     ComputeConceptsFrom(($C, D$), $j + 1$)
```

Figure 4.6, CbO-FC-DBF algorithm pseudo code

The implementation of CbO-FC-DBF is given in *Figure 4.7*. Line numbers 11 to 23 correspond directly to the CbO-FC-DBF algorithm given in *Figure 4.6*. `TQueue` listed in line 6 represents the Queue data structure used in the implementation.

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize, VECTOR_TYPE *BBit,
                         int y) {
    int C[OBJECTSIZE];
    short int D[ATTRIBUTESIZE];
    int CSize, DSize;
    VECTOR_TYPE DBit[VECTOR_MAX_COLS_CELLS];
    TQueue q;
    // Print Concept
    conceptno++;
    int concept = conceptno; // keeps track of the current concept
    int j;
    for (j=y; j<n; j++) {
       if (! isMember(j,BBit)) {
          aIntersectionColj(C,CSize,A,ASize,j);
          deriveIntents(D, DSize, DBit, C, CSize);
          if (isBequalToDuptojBit(BBit, DBit,j)) {
             put(q,C,CSize,D, DSize,DBit,j);
          }
       }
    }
    while (get(q,C,CSize,D,DSize, DBit, j)) {
       ComputeConceptsFrom(C,CSize, D,DSize, DBit, j+1, MBit);
    }
}
```

Figure 4.7, Listing of CbO-FC-DBF program code

Next we will consder the effects of introducing the inherited cannocity test failure enhancement to the basic CbO-FC-DF algorithm. The key concept here is to capture details of intents that fail the cannocity tests and to prevent recomputation of such intents in the next level of recursion. This feature was originally introduced by Krajca et al. (2010) in their FCbO algorithm which also uses a breadth first search stratergy.

The new CbO-FC-ICF-DF (CbO – Full Closure, Inherited Cannocity Test Failure, Depth First Search) algorithm presented in *Figure 4.8* extends the CbO-FC-DF algorithm by incorporating the inherited cannocity test failure stratergy.

4.2.4 CbO-FC-ICF-DF (CbO – Full Closure, Inherited Canonicity Test Failure, Depth First Search Algorithm)

```
CbO-FC-ICF-DF
ComputeConceptsFrom($(A, B), y, \{N^y \mid y \in Y\}$)
1  ProcessConcept($(A, B)$)
2  for $j \leftarrow y$ upto $n - 1$ do
3      $M^j \leftarrow N^j$
4      if $j \notin B$ and $N^j \cap Y_j \subseteq B \cap Y_j$ then
5          $C \leftarrow A \cap \{j\}^{\downarrow}$
6          $D \leftarrow C^{\uparrow}$
7          if $B \cap Y_j = D \cap Y_j$ then
8              ComputeConceptsFrom($(C, D), j + 1, \{M^y \mid y \in Y\}$)
9          else
10             $M^j \leftarrow D$
```

Figure 4.8, CbO-FC-ICF-DF algorithm pseudo code

M^j and N^j are used to capture the intent of failed cannocity tests in the algorithm. Initially M^j is set to the intent of the previously failed cannocity test N^j in line 3. If there are failed cannocity tests at attribute level j, the value of D is captured in M^j in line 10. This is passed to the algorithm during the recursion call as parameter N^j. The key feature of this algorithm is the inheritence of failed cannocity tests. The failed cannocity test is checked at the next level of recursion in line 4 using the following statement.

$$N^j \cap Y_j \subseteq B \cap Y_j$$

In essence the cannoncity test in line 7 and the computation of C and D (in lines 5 and 6) is avoided in situations where the cannocity test has failed before. The implementation of CbO-FC-ICF-DF is given in *Figure 4.9*. Lines 11 to 26 correspond directly to the CbO-FC-ICF-DF algorithm given in *Figure 4.8*. The parameter N^j is represented by an array of bit array pointers. In the code N^j is represented as `VECTOR_TYPE *NBit[]` in line 1. Where `VECTOR_TYPE` represents an unsigned 64 bit integer in this implementation. The function call `isNjSubSetofBuptoJBit()` in line 14 of the code corresponds to the inherited canonicity failure test $N^j \cap Y_j \subseteq B \cap Y_j$ which is given in line 4 of the pseudo code.

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize,
                             VECTOR_TYPE *BBit, int y,  VECTOR_TYPE *NBit[]) {
   int C[OBJECTSIZE];
   short int D[ATTRIBUTESIZE];
   int CSize, DSize;
   VECTOR_TYPE DBit[VECTOR_MAX_COLS_CELLS];
   VECTOR_TYPE *MBit[ATTRIBUTESIZE];
   // Print Concept
   conceptno++;
   int concept = conceptno; // keeps track of the current concept
   int j;
   for (j=y; j<n; j++) {
      CopyRowArrMN(&MBit[j],NBit[j]);
      if (! isMember(j,BBit)) {
          if (isNjSubSetofBuptoJBit(NBit[j],BBit,j)) {

             aIntersectionColj(C,CSize,A,ASize,j);
             deriveIntents(D, DSize, DBit, C, CSize);
             if (isBequalToDuptojBit(BBit, DBit,j)) {
                ComputeConceptsFrom(C,CSize, D,DSize, DBit, j+1, MBit);
             }
             else {
                copyRowArrMN(&MBit[j],DBit);
             }
          }
      }
   }
}
```

Figure 4.9, Listing of CbO-FC-ICF-DF program code

Next we will consider combining the three enhancements. The combination of adding partial closure with incremental closure of intents with the combined depth and breadth first searching features results in the CbO-PC-DBF algorithm (CbO – Partial Closure, Combined Depth First and Breadth First Search) which is presented in *Figure 4.10*. This is a redefinition of the In-Close2 algorithm originally described by Andrews (2011). Due to the incremental closure of intents the introduction of the queue enables a feature called attribute inheritence which allows attributes of the parents intents to be passed to the next level. Due to the breadth first approach to recursion used here it is possible to skip attributes in the current intent, hence the reintroduction of $j \notin B$ in line 2. The implementation of CbO-PC-DBF is given in in Appendix A, *Figure A.1*. Line numbers 10 to 30 correspond directly to the CbO-PC-DBF algorithm given in *Figure 4.10*

4.2.5 CbO-PC-DBF (CbO – Partial Closure, Combined Depth First and Breadth First Search Algorithm)

```
CbO-PC-DBF (In-Close2)
ComputeConceptsFrom(($A, B$), $y$)

1  for $j \leftarrow y$ upto $n-1$ do
2      if $j \notin B$ then
3          $C \leftarrow A \cap \{j\}^{\downarrow}$
4          if $A = C$ then
5              $B \leftarrow B \cup \{j\}$
6          else
7              if $B \cap Y_j = C^{\uparrow Y_j}$ then
8                  PutInQueue($C, j$)
9  ProcessConcept(($A, B$))
10 while GetFromQueue($C, j$) do
11     $D \leftarrow B \cup \{j\}$
12     ComputeConceptsFrom(($C, D$), $j+1$)
```

Figure 4.10, CbO-PC-DBF (In-Close2) algorithm pseudo code (Andrews, 2011)

The combination of adding partial closure with incremental closure of intents with inherited canonicity test failure results in the new CbO-PC-ICF-DF (CbO – Partial Closure, Inherited Cannocity Test Failure, Depth First Search) algorithm is presented in *Figure 4.11*. Here the partial closure of C is stored in M^j when there are failed cannocity tests in line 12. The implementation of the algorithm is given in Appendix *Figure A.2* Lines 12 to 35 of the implementation directly correspond to the CbO-PC-ICF-DF algorithm given in *Figure 4.11*.

4.2.6 CbO-PC-ICF-DF (CbO – Partial Closure, Inherited Canonicity Test Failure, Depth First Search Algorithm)

```
CbO-PC-ICF-DF
ComputeConceptsFrom($(A, B), y, \{N^y \mid y \in Y\}$)

for $j \leftarrow y$ upto $n - 1$ do
    $M^j \leftarrow N^j$
    if $j \notin B$ and $N^j \cap Y_j \subseteq B \cap Y_j$ then
        $C \leftarrow A \cap \{j\}^{\downarrow}$
        if $A = C$ then
            $B \leftarrow B \cup \{j\}$
        else
            if $B \cap Y_j = C^{\uparrow Y_j}$ then
                $D \leftarrow B \cup \{j\}$
                ComputeConceptsFrom($(C, D), j + 1, \{M^y \mid y \in Y\}$)
            else
                $M^j \leftarrow C^{\uparrow Y_j}$
```

Figure 4.11, CbO-PC-ICF-DF algorithm pseudo code

We will next consider an algorithm that combines the inherited cannocity test failure with depth and breadth first search to the CbO-FC-DF algorithm. The resulting CbO-FC-ICF-DBF (CbO – Full Closure, Inherited Cannocity Test Failure, Combined Depth First and Breadth First Search) algorithm presented in *Figure 4.12* is a redefinition of the FCbO algorithm originally described by Krajca et al. (2010).

4.2.7 CbO-FC-ICF-DBF (CbO – Full Closure, Inherited Canonicity Test Failure, Combined Depth First and Breadth First Search Algorithm)

```
CbO-FC-ICF-DBF (FCbO)
ComputeConceptsFrom($(A, B), y, \{N^y \mid y \in Y\}$)
1  ProcessConcept($(A, B)$)
2  for $j \leftarrow y$ upto $n - 1$ do
3      $M^j \leftarrow N^j$
4      if $j \notin B$ and $N^j \cap Y_j \subseteq B \cap Y_j$ then
5          $C \leftarrow A \cap \{j\}^{\downarrow}$
6          $D \leftarrow C^{\uparrow}$
7          if $B \cap Y_j = D \cap Y_j$ then
8              PutInQueue $((C, D), j)$
9          else
10             $M^j \leftarrow D$
11 while GetFromQueue($(C, D), j$) do
12     ComputeConceptsFrom($(C, D), j + 1, \{M^y \mid y \in Y\}$)
```

Figure 4.12, CbO-FC-ICF-DBF (FCbO) algorithm pseudo code (Krajca, Outrata, & Vychodil, 2010)

The code listing in the Appendix A, *Figure A.3* shows the implementation of the CBO-FC-ICF-DBF algorithm. The inherited canonocity test failure and the use of the queue is shown in line 4 and 8 of the pseudo code (*Figure 4.12*)

Finally lets consider an algorithm that combine all three features. *Figure 4.13 ,* lists the CbO-PC-ICF-DBF (CbO – Partial Closure, Inherited Cannocity Test Failure, combined Depth First and Breadth First Search). This is essentially Andrew's In-Close3 algorithm. This incorporates a best of breeds approach by incorporating the partial closures with incremental closure of intents, depth and breadth search and inherited canonicity test failures. The implementation of the CbO-PC-ICF-DBF algorithm is given in Appendix A, *Figure A.4,* Lines 10 to 36 of the implementation directly correspond to the algorithm.

4.2.8 CbO-PC-ICF-BF (CbO – Partial Closure, Inherited Canonicity Test Failure, Combined Depth First and Breadth First Search Algorithm)

```
CbO-PC-ICF-DBF (In-Close3)
ComputeConceptsFrom($(A, B), y, \{N^y \mid y \in Y\}$)

for $j \leftarrow y$ upto $n-1$ do
    $M^j \leftarrow N^j$
    if $j \notin B$ and $N^j \cap Y_j \subseteq B \cap Y_j$ then
        $C \leftarrow A \cap \{j\}^{\downarrow}$
        if $A = C$ then
            $B \leftarrow B \cup \{j\}$
        else
            if $B \cap Y_j = C^{\uparrow Y_j}$ then
                PutInQueue($C, j$)
            else
                $M^j \leftarrow C^{\uparrow Y_j}$
ProcessConcept($(A, B)$)
while GetFromQueue($C, j$) do
    $D \leftarrow B \cup \{j\}$
    ComputeConceptsFrom($(C, D), j+1, \{M^y \mid y \in Y\}$)
```

Figure 4.13, CbO-PC-ICF-BF (In-Close3) algorithm pseudo code (Andrews, 2014)

4.3 Implementation Details

In essence each of the eight algorithms was built up by assembling blocks out of a library of reusable functions (See *Table 4.2*) this ensured that the implementations were carried out in a level playing field. Each of the code blocks were implemented as C++ functions. The implementation of the canonicity test $B \cap Y_j = D \cap Y_j$ is taken as an example to illustrate the implementation of the common block code blocks. The canonicity test given above is used in CbO and FCbO algorithms. This was implemented at code level using the function `isBequalToDuptojBit()` which is listed in *Figure 4.14*. The condition in line 2 checks if `j` is zero. Lines 13-19 of the code makes use of bitwise operations to compare the intents `BBit` and `DBit` which are stored in binary format. The variable `mask` is used to filter this operation only upto the jth element. Lines 8-11 are used to setup the variable `mask`. Variables appearing in uppercase are macros representing constant values used for the bitwise operations specific to the platform used. For instance `VEC_MAX`, `VECTOR_1` and `VECTOR_SIZE2POW` have the values `0xFFFFFFFFFFFFFFFF`, `1i64` and `6` which are specific for a 64bit Windows platform using the Microsoft and Intel compilers. The details of the common data structures used are given in

Table 4.3. The context and the intents were stored in binary format making use of bitwise operations on 64 bit integer arrays. The data type `VECTOR_TYPE` which appears in the implementation code is essentially a macro to an unsigned 64 bit integer. The context was represented as a two dimensional array and the intents as a single dimensional array. The extents were stored as integers. The intents were also duplicated in integer format to facilitate storage of the generated intents. A simple integer based arrays were used in the algorithms that used full closures. Algorithms that used partial closures used a BTree structure to store the integer values of the intents. This was to accommodate the incremental generation of intents. This BTree structure was used only for storing the generated intents and was not used for any computations. In all algorithms computations involving intents were carried out only using the bit array representations. The implementation of algorithms which used a combined depth first and breadth first search strategy used the same queue data structure which appears as `TQueue` in the implementation code, and the ones that used the inherited canonicity test failure as a feature used an array of bit array pointers as the data structure (`VECTOR_TYPE *NBit[]` in the code).

Table 4.3, Main data structures used in the implementations

Implementation	**Data structures used for extents**	**Data structure used for intents**	**Data structures used for context**	**Additional Data Structures**
CbO-FC-DF	integer array	bit array	2D bit array	-
CbO-PC-DF	integer array	bit array	2D bit array	-
CbO-FC-DBF	integer array	bit array	2D bit array	Queue
CbO-PC-DBF	integer array	bit array	2D bit array	Queue
CbO-FC-ICF-DF	integer array	bit array	2D bit array	array of bit array pointers
CbO-PC-ICF-DF	integer array	bit array	2D bit array	array of bit array pointers
CbO-FC-ICF-DBF	integer array	bit array	2D bit array	Queue and an array of bit array pointers
CbO-PC-ICF-BF	integer array	bit array	2D bit array	Queue and an array of bit array pointers

```
bool isBequalToDuptojBit(VECTOR_TYPE *BBit, VECTOR_TYPE *DBit, int j) {
   if (j==0) { // Special case
      return true;
   }
   VECTOR_TYPE mask[VECTOR_MAX_COLS_CELLS];
   VECTOR_TYPE result;
   int pos = j>>VECTOR_SIZE2POW;
   mask[pos] = (VECTOR_1 << j)-1;  // set the current mask element to the correct value

   for (int r = pos-1; r >=0; r--)
      mask[r] = VEC_MAX;  // pad trailing masks with 1

   for (int r=pos; r>=0; r--) {
      result = (BBit[r] ^ DBit[r]) & mask[r];
      if (result != 0)
         return false;
   }
   return true;
}
```

Figure 4.14, Listing of function `isBequalToDuptoj()`

Table 4.4, Test results for real world datasets (time in seconds) with 95% confidence levels

Data Set	**Mushroom** (Frank & Asuncion, 2011)	**Adult**	**Internet Ads**
|G| x |M|	8,124x125	32,561x99	3279x1565
Density	17.36%	11.29%	0.97%
# Concepts	226,920	80,332	16570
CbO-FC-DF (CbO)	1.026 ± 0.0014	0.428 ± 0.0016	1.205 ± 0.0007
CbO-PC-DF (In-Close)	0.866 ± 0.0001	0.357 ± 0.0000	0.220 ± 0.0000
CbO-FC-DBF	1.042 ± 0.0001	0.430 ± 0.0000	1.209 ± 0.0009
CbO-PC-DBF (In-Close2)	0.854 ± 0.0002	0.359 ± 0.0009	0.232 ± 0.0002
CbO-FC-ICF-DF	1.067 ± 0.0002	0.437 ± 0.0002	1.536 ± 0.0006
CbO-PC-ICF-DF	0.461± 0.0010	0.398 ± 0.0011	0.461 ± 0.0011
CbO-FC-ICF-DBF (FCbO)	0.301± 0.0003	0.196 ± 0.0004	0.308 ± 0.0003
CbO-PC-ICF-DBF (In-Close3)	0.253 ± 0.0002	0.167 ± 0.0000	0.166 ± 0.0001

4.4 Empirical Results

4.4.1 Introduction

The algorithms were implemented in C++ and compiled on a Linux Intel C++ Compiler version 15 in debug mode without any optimization (the compiler flags `-g -O0` were used). All the algorithms were implemented in a similar way. The implemented algorithms were tested on the Intel ® AI Cloud running in a Colfax Cluster on a Compute Node that uses an Intel® Xeon ® Gold 6128 @ 3.7 GHz, six core server with 96GB RAM, running a stripped down version of the SUSE Linux Enterprise Server 11 - 64 bit operating system. Section 2.6 of the Methodology Chapter contains the details of the methodology used in carrying out these experiemnets. *Table 4.4* summarizes results of real world datasets with 95% confidence level intervals. In general we can see the following relationships for the real world datasets CbO-PC-ICF-DBF < CbO-FC-ICF-DBF <CbO-PC-DBF < CbO-PC-DF. However CbO-FC-ICF-DBF is placed fourth in the Internet Ad dataset. CbO-PC-ICF-DBF which incorporates all the three features gets the best results for all three datasets.

Table 4.5 lists the detail results for artificial datasets where the density is a variable changing from 25 to 50. The attributes and objects were both fixed at 100 for this experiment. In general we can see the following relationship CbO-PC-ICF-DBF < CbO-PC-DF < CbO-PC-DBF < CBO-PC-ICF-DF < CbO-FC-ICF-DBF < CbO-FC-ICF-DF < CbO-FC-DBF < CbO-FC-DF.

Table 4.6 lists the detail results for artificial datasets where the values of the object were changed from 10,000 to 100,000. The attributes and density were fixed at 100 and 5% respectively for this experiment. Here we observe the following relationship CbO-PC-DF < CbO-PC-DBF < CbO-PC-ICF-DBF < CBO-PC-ICF-DF < CbO-FC-ICF-DBF < CbO-FC-DF < CbO-FC-DBF < CbO-FC-ICF-DF. Here CbO-PC-DF performs better than CbO-PC-ICF-DBF.

Table 4.5, Results for artificial datasets where density is a variable,
|M| = 100 and |G| = 100 (timing in seconds)

Density	Concepts	CbO.FC.DF	CbO.PC.DF	CbO.FC.ICF.DF	CbO.PC.DBF	CBO.FC.DBF	CbO.PC.ICF.DF	CbO.FC.ICF.DBF	CbO.PC.ICF.DBF
25	44,834	0.172	0.036	0.174	0.037	0.177	0.050	0.078	0.029
28	76,366	0.319	0.066	0.321	0.068	0.325	0.090	0.133	0.050
30	120,555	0.499	0.104	0.501	0.106	0.508	0.138	0.205	0.076
33	232,966	1.104	0.224	1.093	0.228	1.112	0.289	0.403	0.151
35	342,210	1.476	0.303	1.463	0.307	1.488	0.391	0.555	0.206
38	703,002	3.533	0.707	3.476	0.719	3.526	0.901	1.146	0.438
40	1,082,618	5.074	1.022	4.994	1.040	5.075	1.295	1.632	0.625
43	2,275,996	11.789	2.330	11.569	2.378	11.688	2.922	3.305	1.318
45	3,913,430	20.137	4.012	19.832	4.093	20.126	4.992	5.373	2.171
48	8,524,316	41.653	8.782	42.317	8.956	42.475	10.874	11.053	4.574
50	14,709,875	75.121	15.145	74.413	15.496	74.536	18.799	18.186	7.624

Table 4.6, Results for artificial datasets where |G| is a variable, |M| = 100 (timing in seconds) and Density = 5%

Objects	Concepts	CbO.FC.DF	CbO.PC.DF	CbO.FC.ICF.DF	CbO.PC.DBF	CBO.FC.DBF	CbO.PC.ICF.DF	CbO.FC.ICF.DBF	CbO.PC.ICF.DBF
10,000	125,304	0.185	0.065	0.197	0.069	0.246	0.098	0.165	0.085
20,000	254,304	0.391	0.139	0.415	0.147	0.423	0.201	0.361	0.181
30,000	374,438	0.589	0.213	0.624	0.230	0.633	0.304	0.520	0.256
40,000	511,907	0.797	0.294	0.849	0.315	0.851	0.413	0.677	0.333
50,000	666,102	1.015	0.378	1.089	0.399	1.071	0.535	0.844	0.418
60,000	824,485	1.229	0.464	1.319	0.484	1.294	0.660	1.014	0.505
70,000	1,002,296	1.462	0.556	1.577	0.577	1.530	0.804	1.203	0.611
80,000	1,193,271	1.712	0.657	1.844	0.675	1.787	0.931	1.422	0.726
90,000	1,385,125	1.958	0.760	2.113	0.776	2.023	1.085	1.633	0.851
100,000	1,581,947	2.218	0.862	2.399	0.897	2.317	1.241	1.871	0.983

The detail results of varying the attributes from 1000 to 2000 while keeping the objects and the density fixed at 100 and 5% respectively are given in *Table 4.7*. Here we observe the following relationship in general CbO-PC-ICF-DBF < CbO-PC-DF < CbO-PC-DBF < CBO-PC-ICF-DF < CbO-FC-ICF-DBF < CbO-FC-ICF-DF < CbO-FC-DF < CbO-FC-DBF.

Table 4.7, Results for artificial datasets where |M| is a variable, |G| = 100 (timing in seconds) and Density = 5%

Attribs.	**Concepts**	**CbO.FC. DF**	**CbO.PC .DF**	**CbO.FC. ICF.DF**	**CbO.PC. DBF**	**CBO.FC. DBF**	**CbO.PC. ICF.DF**	**CbO.FC. ICF.DBF**	**CbO.PC. ICF.DBF**
1,000	7,801	1.073	0.068	0.806	0.071	1.143	0.150	0.321	0.063
1,100	8,690	1.483	0.091	1.077	0.101	1.577	0.204	0.431	0.082
1,200	9,277	1.894	0.111	1.417	0.114	2.032	0.256	0.548	0.102
1,300	10,086	2.414	0.130	1.792	0.142	2.560	0.309	0.690	0.120
1,400	10,914	3.066	0.157	2.250	0.171	3.265	0.378	0.856	0.142
1,500	12,026	3.905	0.190	2.830	0.205	4.130	0.465	1.051	0.170
1,600	12,529	4.532	0.218	3.389	0.228	4.826	0.549	1.250	0.198
1,700	13,319	5.572	0.251	4.020	0.267	5.897	0.658	1.508	0.234
1,800	14,263	6.587	0.282	4.813	0.296	6.994	0.761	1.718	0.266
1,900	15,001	7.859	0.327	5.577	0.344	8.259	0.858	1.998	0.301
2,000	16,577	9.683	0.391	6.765	0.399	10.244	1.008	2.434	0.354

One interesting observation is that CbO-FC-ICF-DBF performs poorly in the artificial datasets compared to the real datasets. This could be due to the randomness in the artificial datasets.

4.4.2 Statistical Significance of the empirical results

The results obtained in *Table 4.4*, *Table 4.5*, *Table 4.6* and *Table 4.7* were checked to see if they had statistical significance by first carrying out one-way analysis of variance (one way Anova). For the latter three tables the largest dataset used was analyzed for statistical significance. Specifically, the three datasets considered from *Table 4.5*, *Table 4.6* and *Table 4.7* are n100m100d50s1000 *(density = 50)*, n100m100000d5s1000 *(objects = 100,000)* and n2000m100d5s1000 *(attributes = 2000)* respectively.

The null hypothesis in one-way analysis H_0 is that the means of the empirical timing of all the eight algorithms for a given dataset are the same. We can clearly see that the conditions $F > F_{crit}$ and $P_{value} < \alpha$ is valid for all six datasets that were analyzed (See *Table 4.8*). Here α was taken as 0.05.

Table 4.8 shows a summary of the one-way analysis. Hence according to one-way analysis we can reject H_0 the null hypothesis. We can conclude that for each dataset there exists at least one algorithm that has a mean which is statistically different from others. Section 2.3.2 shows how the one-way analysis was carried out.

Table 4.8, One-way Analysis of the major datasets used in the analysis

dataset	F	F_{crit}	P_{value}
ad	3,755,529.9	2.1397	1.49E-197
mushroom	24,113.0	2.1397	1.24E-118
adult	132,049.8	2.1397	3.26E-145
n2000m100d5s1000	10,829,084.8	2.1397	4.13E-214
n100m100d50s1000	151,826,683.3	2.1397	2.15E-255
n100m100000d5s1000	402.1	2.1397	4.82E-55

Next to show that the means of the top two algorithms for each dataset has statistical significance the t-Test for two sample assuming unequal variances analysis was carried out. *Table 4.9*, shows a summary of the t-Test.

Table 4.9, t-Test for two sample assuming unequal variances analysis of the major datasets used in the analysis

dataset	t_{stat}	mean1	mean2	$t_{critical\ two\ tail}$	$P_{two\text{-}tail}$	algo1	algo2
ad	1004.02	0.2197	0.1664	2.1788	6.42E-31	In-Close	In-Close3
mushroom	-539.98	0.2531	0.3013	2.1788	1.10E-27	In-Close3	FCbO
adult	-144.07	0.1674	0.1961	2.2622	1.90E-16	In-Close3	FCbO
n2000m100d5s1000	-130.33	0.3538	0.3907	2.1448	5.39E-23	In-Close3	In-Close
n100m100d50s1000	-8276.50	7.6245	15.1448	2.1098	1.37E-57	In-Close3	In-Close
n100m100000d5s1000	-1.86	0.7147	0.8969	2.2622	9.64E-02	In-Close	In-Close2

The null hypothesis in the t-Test analysis H_0 is that the mean values of the two fastest algorithms considered are the same. For first five datasets in *Table 4.9* we can clearly see that $|t_{stat}| > t_{critical\ two-tail}$ and $P_{two-tail} < \alpha$ is valid. Here α was taken as 0.05. Here based on the t-Test we can reject H_0 the null hypothesis. Hence, we can conclude that In-Close3 is the fastest algorithm for the datasets ad, mushroom, adult, n2000 and d50. For the last dataset n100m100000d5s1000, both the conditions are invalid. For the random dataset results shown in *Table 4.6* where we increased the number of objects In-Close (CbO-PC-DF) and the In-Close2 (CbO-PC-DBF) algorithms produced better results than In-Close3. This was the only dataset series where In-Close3 wasn't the fastest. In-Close3 was the fastest for all the real-world datasets that were considered as well. Section 2.3.3 shows how the t-Test for two sample assuming unequal variances analysis was carried out.

4.4.3 Impact of the three enhancements in isolation

4.4.3.1 Introduction

Let's initially consider the impact of the three enhancements in isolation. *Figures 4.15* to *Figure 4.21* are used to compare algorithms two at a time to show the impact of the three enhancements.

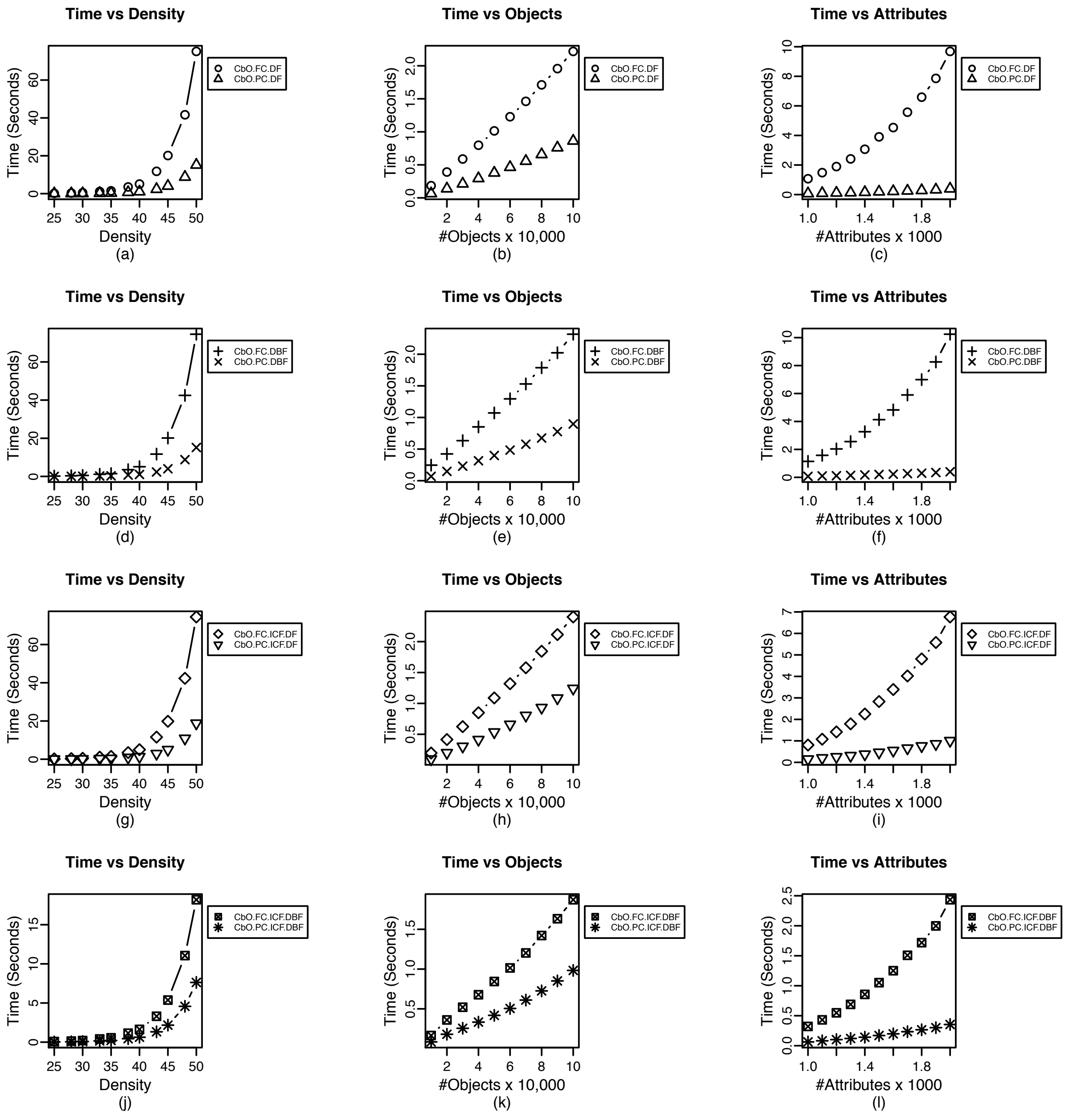


Figure 4.15, Highlighting the effects of Partial Closures by comparing the results of CbO-FC-DF vs CbO-PC-DF, CbO-FC-DBF vs CbO-PC-DBF, CbO-FC-ICF-DF vs CbO-PC-ICF-DF and CbO-FC-ICF-BDF vs CbO-PC-ICF-BDF

Due to the significant variation in the timing results of the eight algorithms considered in *Tables 4.5*, *Table 4.6*, *Table 4.7* it is difficult to represent these results using only three separate graphs. The comparison of experimental results between two carefully selected algorithms will help in the better interpretation of the empirical results obtained.

4.4.3.2 The impact of using partial closures with incremental closure of intents

To examine the impact of using partial closures with incremental closure of intents in CbO based algorithms we can compare the results of the CbO-FC-DF algorithm vs CbO-PC-DF algorithm, CbO-FC-DBF algorithm vs CbO-PC-DBF algorithm, CbO-FC-ICF-DF algorithm vs CbO-PC-ICF-DF algorithm and the CbO-FC-ICF-BDF algorithm vs CbO-PC-ICF-BDF algorithm. In each of the four cases the only difference between the considered algorithms is that one of the algorithm uses full closures and the other uses partial closures with incremental closure of intents.

Figure 4.15 shows a detail comparison of these four sets of algorithms for the artificial datasets where the density, objects and attributes are varied respectively. The details values shown in the graphs are presented in *Table 4.6* and *Table 4.7* respectively. *Figure 4.15(a)(b)(c)* compares the CbO-FC-DF algorithm and the CbO-PC-DF algorithm for variations of Density, Objects and Attributes respectively. The symbols ○ and △ are used to represent the algorithms CbO-FC-DF and CbO-PC-DF respectively in the graphs. We can see clearly that CbO-PC-DF performs significantly better compared to CbO-FC-DF in all three cases. *Figure 4.15(d)(e)(f)* compares the CbO-FC-DBF algorithm and the CbO-PC-DBF algorithm for the same variations. Here the symbols + and × are used to represent the algorithms CbO-FC-DBF and CbO-PC-DBF respectively in the graphs. Here the CbO-PC-DBF algorithm performs significantly better compared to CbO-FC-DBF in all three cases. *Figure 4.15(g)(h)(i)* compares the CbO-FC-ICF-DF algorithm and the CbO-PC-ICF-DF algorithm for the three variations. The symbols ◇ and ▽ are used to represent the CbO-FC-ICF-DF and CbO-PC-ICF-DF algorithms respectively in the graphs. CbO-PC-ICF-DF performs significantly better compared to CbO-FC-ICF-DF in all three cases. Finally *Figure 4.15(j)(k)(l)* compares the CbO-FC-ICF-DBF algorithm and the CbO-PC-ICF-DBF algorithm for the same variations. Here the symbols ⊠ and ✳ are used to represent the CbO-FC-ICF-DBF and CbO-PC-ICF-DBF algorithms in the graphs. The CbO-PC-ICF-DBF algorithm is faster compared to CbO-FC-ICF-DBF in all three cases. In the four sets of comparisons we can clearly see the performance improvement in using partial closures with the incremental closure of intents instead of using full closures.

4.4.3.3 The impact of using combined depth first and breadth first search

The impact of the combined depth first and breadth first approach can be observed by comparing the performance of the CbO-FC-DF vs CbO-FC-DBF algorithms, CbO-PC-DF vs CbO-PC-DBF algorithms, CbO-FC-ICF-DF vs CbO-FC-ICF-DBF algorithms and CbO-PC-ICF-DF vs CbO-PC-ICF-DBF algorithms. In each of the comparison one of the algorithms uses a pure depth first approach and the second algorithm uses a combined depth first and breadth first approach. *Figure 4.16* shows a detail comparison of these four sets of algorithms for the artificial datasets where the density, objects and attributes are varied respectively. *Figure 4.16(a)(b)(c)* compares the CbO-FC-DF algorithm and the CbO-FC-DBF algorithm for the three variations. Here we do not see a significant difference between the two algorithms. The same symbols used in *Figure 4.15* are used to represent each algorithm in the remaining graphs that presented. *Figure 4.16(d)(e)(f)*, compares the CbO-PC-DF algorithm and the CbO-PC-DBF algorithm for the same variations. Here too the results are similar. *Figure 4.16(g)(h)(i)* compares the CbO-FC-ICF-DF algorithm and the CbO-FC-ICF-DBF algorithm for the variations. The CbO-FC-ICF-DBF algorithm performs significantly better than the CbO-FC-ICF-DF algorithm. Finally *Figure 4.16(j)(k)(l)* compares the CbO-PC-ICF-DF algorithm and the CbO-PC-ICF-DBF algorithm. In this comparison CbO-PC-ICF-DBF algorithm performs better than the CbO-PC-ICF-DF algorithm.

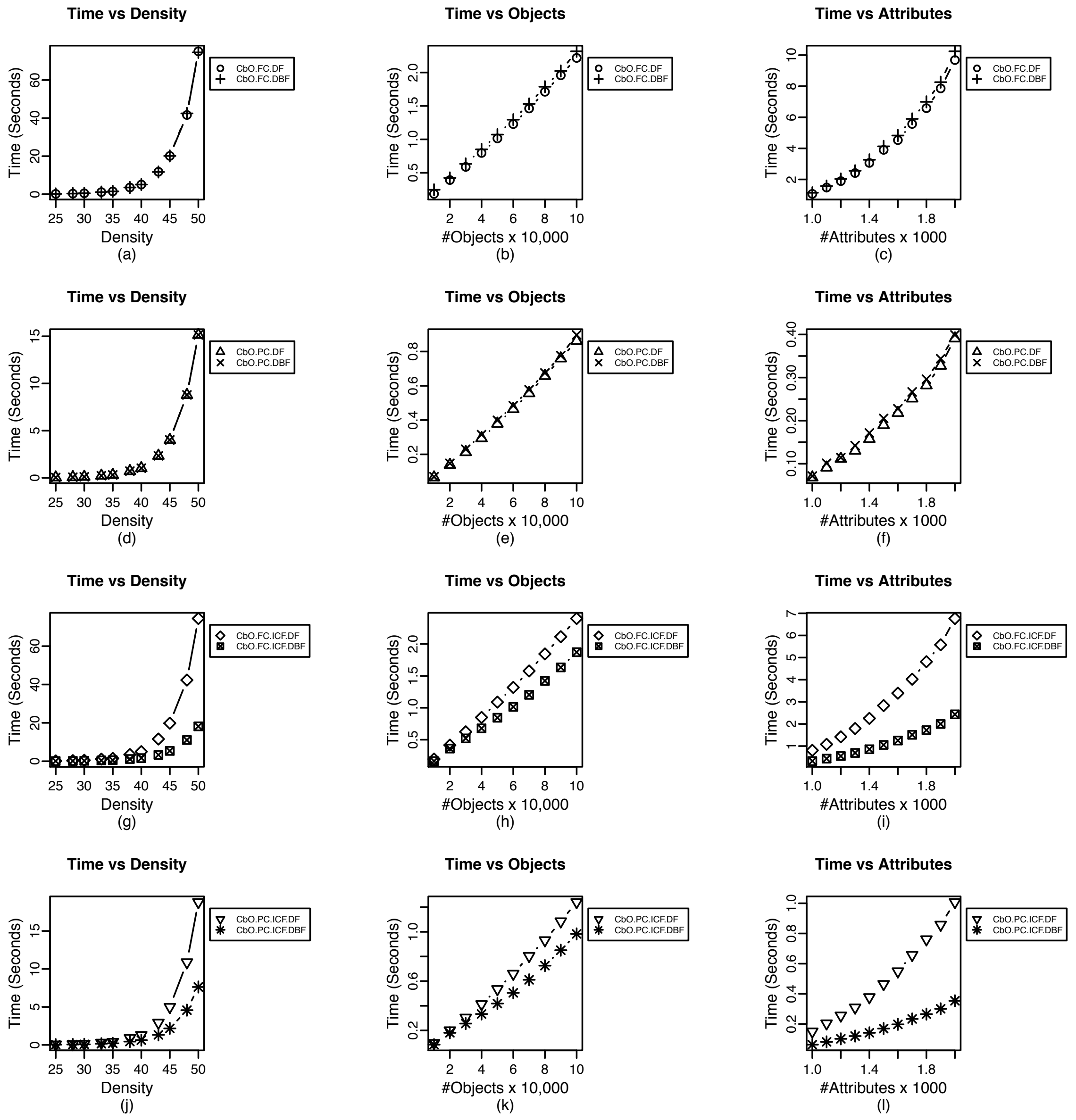


Figure 4.16, Highlighting the effects of Combined Depth First and Breadth First Search by comparing the results of CbO-FC-DF vs CbO-FC-DBF, CbO-PC-DF vs CbO-PC-DBF, CbO-FC-ICF-DF vs CbO-FC-ICF-DBF and CbO-PC-ICF-DF vs CbO-PC-ICF-DBF

These results imply that the impact of the combined depth first and breadth first search feature is significant when used in conjunction with the canonicity test failure feature.

4.4.3.4 The impact of the inherited cannocity test failure

Finally the impact of the inherited canonicity test failure can be determined by comparing the performance of the CbO-FC-DF vs CbO-FC-ICF-DF algorithms, CbO-PC-DF vs CbO-PC-ICF-DF algorithms, CbO-FC-DBF vs CbO-FC-ICF-DBF algorithms and CbO-PC-DBF vs

CbO-PC-ICF-DBF algorithms. In each comparison one algorithm uses the inherited canonicity test failure and the other algorithm does not.

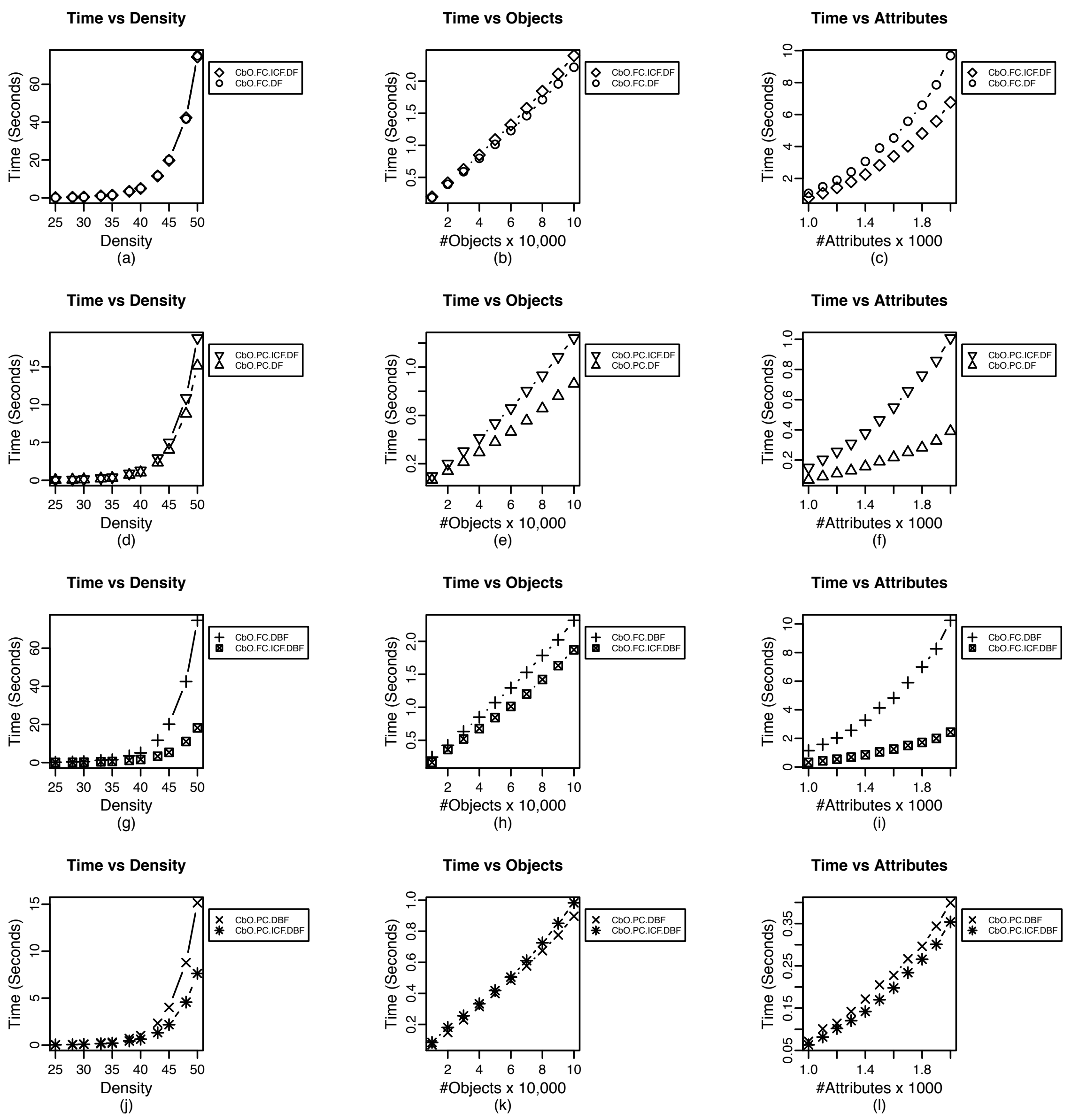


Figure 4.17, Highlighting the effects of the Inherited Canonicity Test Failure by comparing the results of CbO-FC-DF vs CbO-FC-ICF-DF, CbO-PC-DF vs CbO-PC-ICF-DF, CbO-FC-DBF vs CbO-FC-ICF-DBF and CbO-PC-DBF vs CbO-PC-ICF-DBF

Figure 4.17 shows a detail comparison of these four algorithms where density, objects and attributes are varied for random datasets. *Figure 4.17(a)(b)(c)* compares the CbO-FC-DF algorithm and the CbO-FC-ICF-DF algorithm for these three variations. The results are similar for the three graphs. *Figure 4.17(d)(e)(f)* compares the CbO-PC-DF algorithm and the CbO-PC-ICF-DF algorithm. Here we observe that CbO-PC-ICF-DF algorithm performs poorly compared to CbO-PC-DF. We can note that the inherited canonicity test failure doesn't perform well with partial closures used in combination with a depth first search approach. *Figure 4.17(g)(h)(i)* compares the CbO-FC-DBF algorithm and the CbO-FC-ICF-DBF algorithm. In here the CbO-FC-ICF-DBF algorithm performs significantly better than CbO-FC-DBF algorithm. Finally *Figure 4.17(j)(k)(l)* compares the CbO-PC-DBF algorithm and the CbO-PC-ICF-DBF algorithm. Here the CbO-PC-ICF-DBF algorithm performs better than CbO-PC-DBF algorithm.

This is significant for variations of density and attributes. From the comparisons of the graphs shown in *Figure 4.17(g)(h)(i)(j)(k)(l)* and *Figure 4.16(g)(h)(i)(j)(k)(l)* we can conclude that inherited canonicity test failure works well in combination with the combined depth first and breadth first search approach.

4.4.3.5 The impact of the combination of the partial closure with the incremental closure of intents joined with the combined depth first and breadth first search

To see the effects of the three enhancements in combination, let's first consider the effect of partial closures with incremental closures of intents joined with combined depth first and breadth first search. A comparison of the results of the algorithms CbO-FC-DF vs CbO-PC-DBF and CbO-FC-ICF-DF vs CbO-PC-ICF-DBF will allow us to observe the combined effect of these two features. *Figure 4.18* gives a detail comparison between these two sets of algorithms. *Figure 4.18(a)(b)(c)* compares the CbO-FC-DF, CbO-PC-DBF algorithms and *Figure 4.18(d)(e)(f)* compares the CbO-FC-ICF-DF, CbO-PC-ICF-DBF algorithms. In both cases the algorithm which has the combined features (CbO-PC-DBF and CbO-PC-ICF-DBF) performs significantly than the other algorithm. This leads us to conclude that partial closures with incremental closures of intents works well with combined depth first and breadth first search combination.

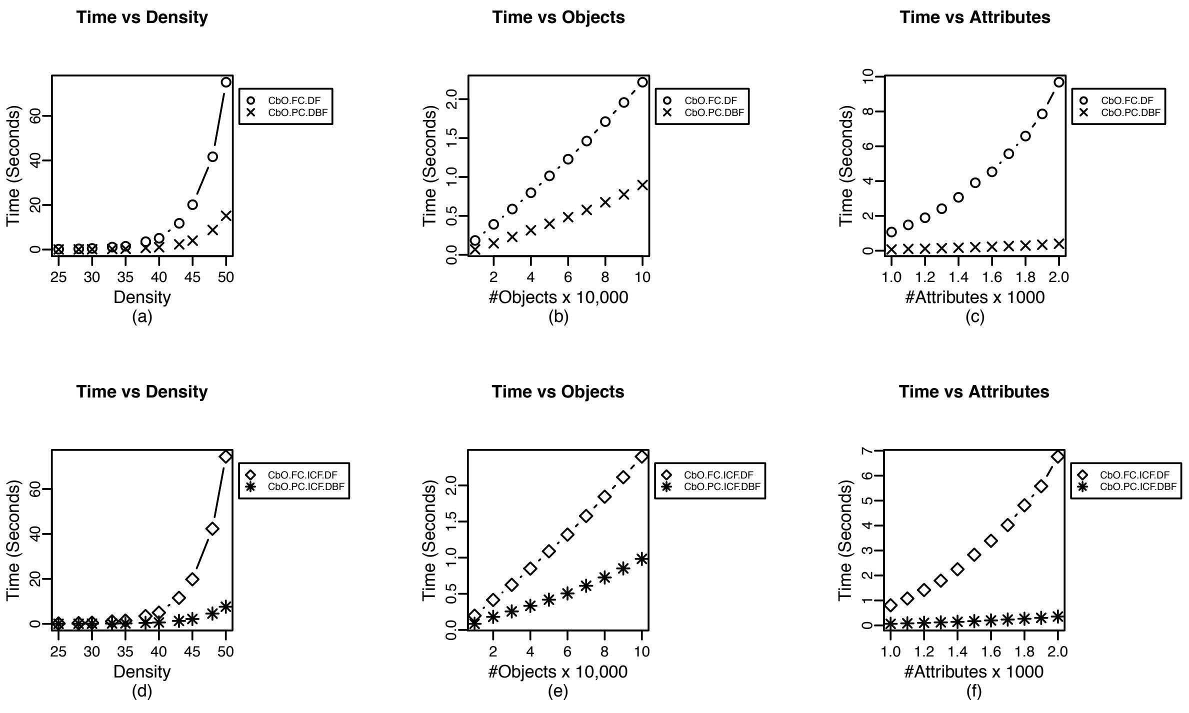


Figure 4.18, Highlighting the effects of Partial Closures with Incremental Closure of Intents and the Combined Depth First and Breadth First approach by comparing the results of CbO-FC-DF vs CbO-PC-DBF and CbO-FC-ICF-DF vs CbO-PC-ICF-DBF

4.4.3.6 The impact of the combination of the combined partial closure with the incremental closure of intents joined with the inherited cannocity test failure

The effects of the combined partial closure with the incremental closure of intents with the inherited canonicity test failure can be examined by comparing the algorithms presented in *Figure 4.19.* The algorithms CbO-FC-DF and CbO-PC-ICF-DF are compared in *Figure 4.19(a)(b)(c).* The CbO-FC-DBF algorithm and the CbO-PC-ICF-DBF algorithm are compared in *Figure 4.19(d)(e)(f).* Both CbO-PC-ICF-DF and CbO-PC-ICF-DBF perform significantly better than the other algorithm. We can conclude that partial closure with the incremental closure of intents works well with inherited canonicity test failures.

4.4.3.7 The impact of the combination of the combined depth first and breadth first search joined with the inherited cannocity test failure

Finally, we can make further observations regarding the effects of using the combined depth first and breadth first search with the inherited canonicity test failure in detail by comparing the algorithms presented in *Figure 4.20.* The algorithms CbO-FC-DF and CbO-FC-ICF-DBF are compared in *Figure 4.20 (a)(b)(c).*

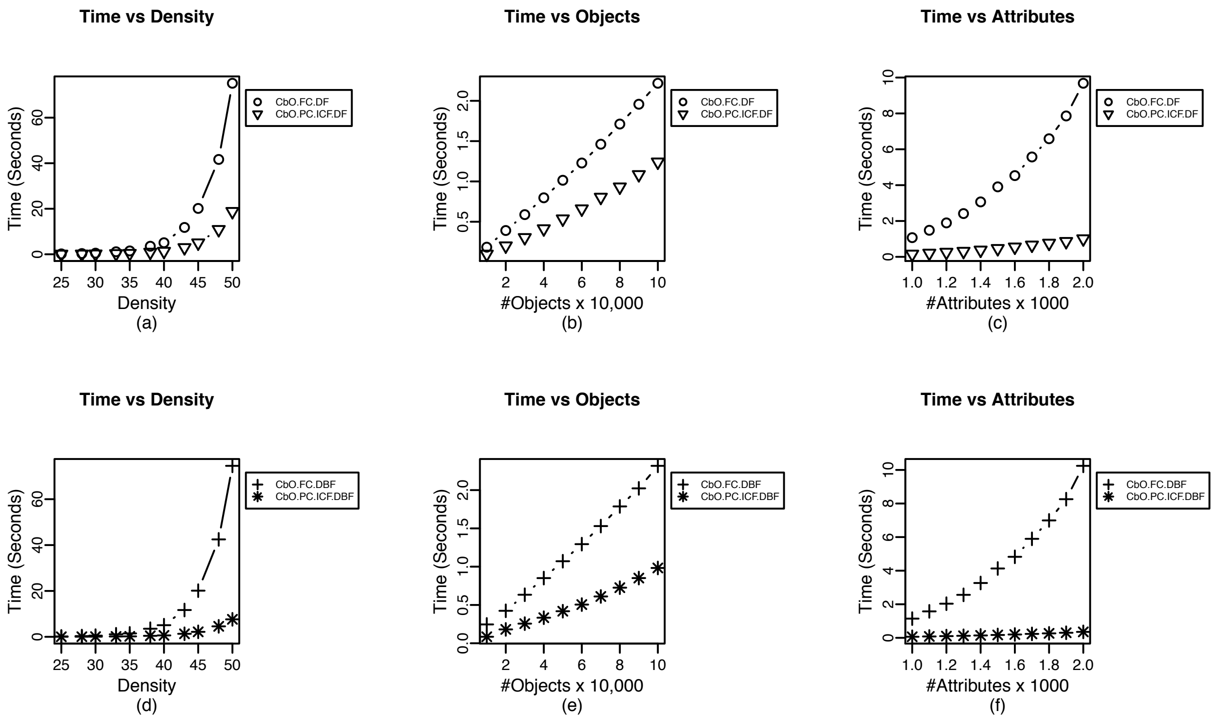


Figure 4.19,Highlighting the effects of Partial Closures combined with the Inherited Canonicity Test Failure by comparing the results of CbO-FC-DF vs CbO-PC-ICF-DF and CbO-FC-DBF vs CbO-PC-ICF-DBF

The CbO-FC-ICF-DBF algorithm performs significantly better than the CbO-FC-DF algorithm. The CbO-PC-DF algorithm and the CbO-PC-ICF-DBF algorithm are compared in the graphs given in *Figure 4.20 (d)(e)(f)*. Here the CbO-PC-ICF-DBF algorithm performs better than the CbO-PC-DF algorithm. For variations of objects (See *Figure (e)*) CbO-PC-ICF-DBF is slower than CbO-PC-DF. These results and the observations made earlier regarding the combined depth first and breadth first search with the inherited canonicity test failure show that performance improvements with this combination is more apparent for full closures as opposed to partial closures.

4.4.3.8 The effect of the combination of all three features

To observe the effect of combining all the three enhancements that are considered we can compare the performance of the algorithms CbO-FC-DF with CbO-PC-ICF-DBF (See *Figure 4.21*). The graphs clearly indicate that CbO-PC-ICF-DBF performs significantly better than CbO-FC-DF.

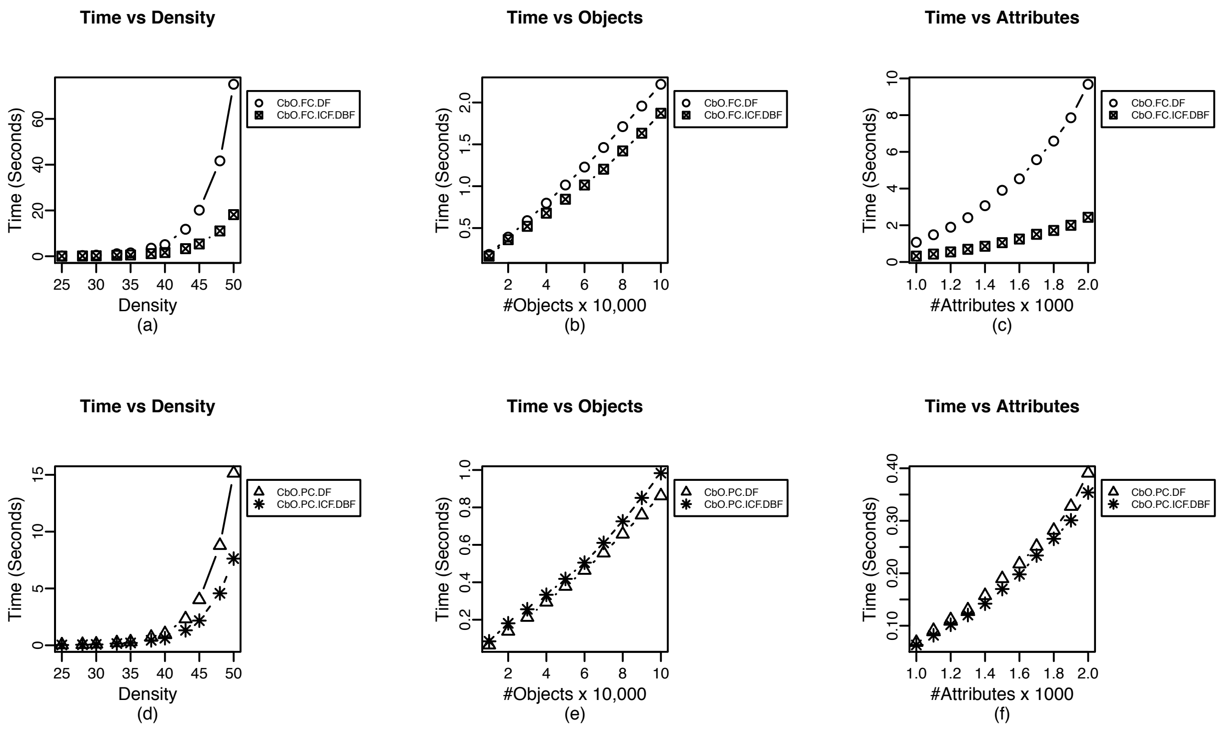


Figure 4.20, Highlighting the effects of Combined Depth First and Breadth First Search used in combination with the Inherited Canonicity Test Failure by comparing the results of CbO-FC-DF vs CbO-FC-ICF-DBF and CbO-PC-DF vs CbO-PC-ICF-DBF

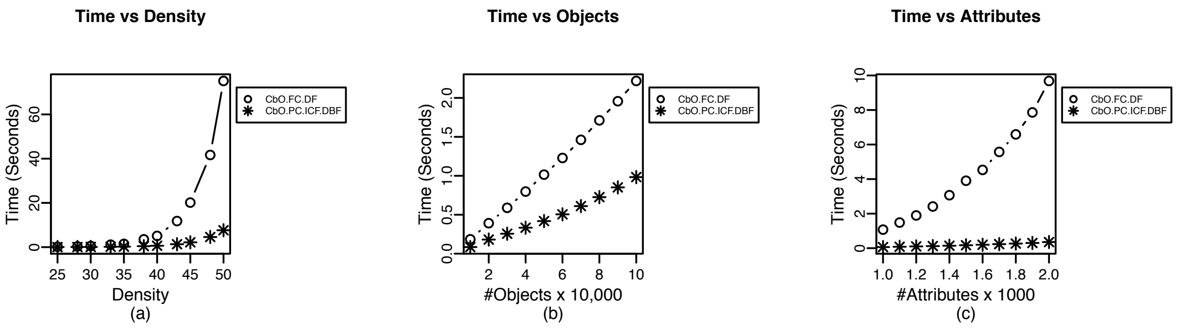


Figure 4.21, Highlighting the effects of combining all three enhancements by comparing CbO-FC-DF with CbO-PC-ICF-DBF

Figure 4.16(a)(b)(c)(d)(e)(f) and *Figure 4.17 (a)(b)(c)(d)(e)(f)* clearly show there is no significant performance enhancement with the combined depth first and breadth first search or the inherited canonicity test are used in isolation.

4.4.4 Analysis of the effects of the three variations

Table 4.10 presents the speedup obtained for the algorithm combinations considered in the graphs for the following datasets Density - n100m100d50s1000 *(density = 50%, #objects =*

100, #attributes = 100), Objects - n100m100000d5s1000 *(#objects = 100,000, #attributes = 100, density = 5%)*, and Attributes – n2000m100d5s1000 *(#attributes = 2,000, #objects = 100, density = 5%)*.

Table 4.10(a)(b)(c)(d)(e)(f)(g)(h) represents each of the seven cases that were considered in the graphs which looked at the effects the of three features partial closures with incremental closure of intents (PC), combined depth and breadth first search (DBF), the inherited canonicity test failure (ICF) and the four combinations that were considered. For instance the speed up 6.71x shown in *Table 4.8(a)* for the third row ICF+DF under the attributes column means that when the attributes were varied for the dataset n2000m100d5s1000.cxt the speedup observed by keeping ICF and Depth First (DF) constant and varying PC only was 6.71x. This corresponds to the ratio of the last values plotted in the graph in *Figure 4.15(i)* for the algorithms CbO-FC-ICF-DF and CbO-PC-ICF-DF. We can clearly see a significant speedup when partial closure with the incremental closure of intents is considered in isolation (See *Table 4.10(a)*). There is in fact a negative impact when the inherited canonicity test failure is considered in isolation keeping partial closure with incremental closure of intents and depth first search constant (See the values less than one in *Table 4.10(c)* row 2 and *Figure 4.17(d)(e)(f)*).

Table 4.10, Speedup of the comparison of algorithms for the Density - n100m100d50s1000.cxt, Objects - n100m100000d5s1000.cxt, Attributes – n2000m100d5s1000.cxt

Partial Closure

	Densi ty	Extents	Intents
DF	4.96	2.57	24.78
DBF	4.81	2.58	25.68
ICF+DF	3.96	1.93	6.71
ICF+PC	2.39	1.90	6.88

(a)

Depth first and Breadth first search

	Density	Extents	Intents
FC	4.85	0.96	0.95
PC	0.98	0.96	0.98
ICF+FC	4.09	1.28	2.78
ICF+PC	2.47	1.26	2.85

(b)

Inherited Cannoccity Test Failure

	Density	Extents	Intents
FC+DF	1.01	0.92	1.43
PC+DF	0.81	0.69	0.39
FC+DBF	4.10	1.24	4.21
PC+DBF	2.03	0.91	1.13

(c)

PC+DBF

	Densi ty	Extents	Intents
*	4.85	2.47	24.27
ICF	9.76	2.44	19.12

(d)

PC+ICF

	Density	Extents	Intents
DF	4.00	1.79	9.61
DBF	9.78	2.36	28.96

(e)

DBF+ICF

	Density	Extents	Intents
FC	4.13	1.19	3.98
PC	7.62	0.88	1.10

(f)

PC+DBF+ICF

	Density	Extents	Intents
*	9.85	2.26	27.37

(g)

This is true even for the instance when full closures and depth first search are kept constant when inherited canonicity test failure is considered in *Table 4.10(c)* row 1 and *Figure 4.17(a)(b)(c)*).

Table 4.10(d)(e) highlights the performance improvements obtained when partial closure with incremental closure of intents is combined respectively with the combined depth first and breadth first search feature and the inherited canonicity failure feature. We do not see the same levels of speedup when we consider the inherited canonicity test failure is combined with the combined depth and breadth first search (See *Table 4.10(f)*). We also can observe that the speedup obtained for partial closures is lower than that of full closures in this instance.

4.4.5 Validating the results of instrumenting closures and intersections

The programs were instrumented to count the number of closures and intersections carried out to get further insights of the timing results obtained earlier. These are presented in

Table 4.11 and *Table 4.12* for real world and artificial datasets respectively. Algorithms that use partial closure with incremental closure of intents have significantly less number of intersections confirming the timing results obtained earlier. The number of intersections for CbO-FC-DBF compared to CbO-FC-DF and CbO-PC-DBF compared to CbO-PC-DF shows that the total number of intersections are almost the same as of the original. This confirms that the combined depth and breadth first search used in isolation has no significant performance improvement. In addition both CbO-FC-ICF-DF and CbO-PC-ICF-DF show a similar number of intersections with their counterparts without the inherited canonicity test failure algorithms (CbO-FC-DF and CbO-PC-DF respectively) confirming that using the inherited canonicity test failure in isolation has no significant performance improvement. Nj Density used in

Table 4.11 and *Table 4.12,* is an instrumentation value that computes the average percentage of inherited canonicity test failures that each recursion level gets from its parent. We can clearly see this percentage is very low for CbO-FC-ICF-DF and CbO-PC-ICF-DF algorithms. However when inherited canonicity test failure is used in combination of the combined depth first and breadth first strategy (CbO-FC-ICF-DBF and CbO-PC-ICF-DBF) the algorithm is able to compute all the inherited canonicity test failures for that level of recursion before sending it to the child recursion. Hence we see a higher percentage Nj density for CbO-FC-ICF-DBF and CbO-PC-ICF-DBF algorithms. This confirms the timing results that were obtained earlier that inherited canonicity test failure gives better performance only when combined with the combined depth and breadth first search feature. The lowest number of intersections in the instrumentation results is for the algorithm CbO-PC-ICF-DBF confirming that there is a significant performance increase when all three features are used in combination.

Table 4.11,Comparison of closures and intersections for the real datasets.

Mushroom	**Full Closure/Partial Closures**	**Intersections**	**Extent Intersections**	**Total Intersections**	**Nj Density (%)**
CbO-FC-DF	3,810,933	476,366,625	3,810,932	480,177,557	-
CbO-PC-DF	3,810,932	398,366,498	4,094,444	402,460,942	-
CbO-FC-DBF	3,810,971	476,371,375	3,810,971	480,182,346	-
CbO-PC-DBF	3,810,932	398,366,498	3,830,027	402,196,525	-
CbO-FC-ICF-DF	3,810,971	476,371,375	3,810,971	480,182,346	17
CbO-PC-ICF-DF	3,810,932	398,366,498	4,094,444	402,460,942	17
CbO-FC-ICF-DBF	402,301	50,287,625	402,301	50,689,926	70
CbO-PC-ICF-DBF	402,295	42,480,969	421,390	42,902,359	69
Adult					
CbO-FC-DF	3,260,451	322,784,649	3,260,450	326,045,099	-
CbO-PC-DF	3,260,450	250,928,223	3,293,914	254,222,137	-
CbO-FC-DBF	3,260,450	322,784,550	3,260,450	326,045,000	-
CbO-PC-DBF	3,260,450	250,928,223	3,279,787	254,208,010	-
CbO-FC-ICF-DF	3,260,450	322,784,550	3,260,450	326,045,000	7
CbO-PC-ICF-DF	3,260,450	250,928,223	3,293,914	254,222,137	7
CbO-FC-ICF-DBF	330,674	32,736,726	330,674	33,067,400	94
CbO-PC-ICF-DBF	330,674	22,004,443	350,011	22,354,454	94
Ad					
CbO-FC-DF	6,307,712	9,871,569,280	6,307,711	9,877,876,991	-
CbO-PC-DF	6,307,711	6,911,051,986	6,355,808	6,917,407,794	-
CbO-FC-DBF	6,307,711	9,871,567,715	6,307,711	9,877,875,426	-
CbO-PC-DBF	6,307,711	6,911,051,986	6,335,635	6,917,387,621	-
CbO-FC-ICF-DF	6,307,711	9,871,567,715	6,307,711	9,877,875,426	29
CbO-PC-ICF-DF	6,307,711	6,911,051,986	6,355,808	6,917,407,794	29
CbO-FC-ICF-DBF	323,540	506,340,100	323,540	506,663,640	96
CbO-PC-ICF-DBF	323,540	376,435,030	351,464	376,786,494	96

Table 4.12, Comparison of closures and intersections for artificial datasets.

n100m50000d5s1000.cxt	**Full Closure/Partial Closures**	**Intersections**	**Extent Intersections**	**Total Intersections**	**Nj Density (%)**
CbO-FC-DF	14,554,321	1,455,432,100	14,554,320	1,469,986,420	-
CbO-PC-DF	14,554,320	1,179,770,310	14,673,051	1,194,443,361	-
CbO-FC-DBF	14,554,320	1,455,432,000	14,554,320	1,469,986,320	-
CbO-PC-DBF	14,554,320	1,179,770,310	14,671,479	1,194,441,789	-
CbO-FC-ICF-DF	14,554,320	1,455,432,000	14,554,320	1,469,986,320	12
CbO-PC-ICF-DF	14,554,320	1,179,770,310	14,673,051	1,194,443,361	12
CbO-FC-ICF-DBF	5,435,185	543,518,500	5,435,185	548,953,685	59
CbO-PC-ICF-DBF	5,435,185	434,151,688	5,552,344	439,704,032	59
n1500m100d5s1000.cxt					
CbO-FC-DF	8,530,332	12,795,498,000	8,530,331	12,804,028,331	-
CbO-PC-DF	8,530,331	8,743,545,457	8,557,266	8,752,102,723	-
CbO-FC-DBF	8,530,331	12,795,496,500	8,530,331	12,804,026,831	-
CbO-PC-DBF	8,530,331	8,743,545,457	8,556,927	8,752,102,384	-
CbO-FC-ICF-DF	8,530,331	12,795,496,500	8,530,331	12,804,026,831	22
CbO-PC-ICF-DF	8,530,331	8,743,545,457	8,557,266	8,752,102,723	22
CbO-FC-ICF-DBF	1,103,074	1,654,611,000	1,103,074	1,655,714,074	87
CbO-PC-ICF-DBF	1,103,074	1,088,468,535	1,129,670	1,089,598,205	87
n100m100d45s1000.cxt					
CbO-FC-DF	77,957,720	7,795,772,000	77,957,719	7,873,729,719	-
CbO-PC-DF	77,957,719	6,425,280,534	80,220,058	6,505,500,592	-
CbO-FC-DBF	77,957,719	7,795,771,900	77,957,719	7,873,729,619	-
CbO-PC-DBF	77,957,719	6,425,280,534	79,550,133	6,504,830,667	-
CbO-FC-ICF-DF	77,957,719	7,795,771,900	77,957,719	7,873,729,619	12
CbO-PC-ICF-DF	77,957,719	6,425,280,534	80,220,058	6,505,500,592	12
CbO-FC-ICF-DBF	17,193,707	1,719,370,700	17,193,707	1,736,564,407	72
CbO-PC-ICF-DBF	17,193,707	1,379,000,449	18,786,121	1,397,786,570	72

4.5 Analysis of the Eight CbO based Algorithms

4.5.1 Introduction

The theoretical complexity of the original CbO algorithm has been shown to be $O(|G|^2|M||L|)$. The eight other variations although improvements, have same theoretical complexity of $O(|G|^2|M||L|)$. A detail complexity analysis of each of these recursive algorithms is a complex task and beyond the scope of this thesis. Instead an analytical comparison between the eight different variations of CbO is presented. The results of the analysis tallies with the experimental results obtained earlier in the chapter.

The eight different variations of the CbO algorithm are made up of 17 common code blocks which are presented in *Table 4.2*. The algorithmic complexity of each block is shown in *Table 4.13* using Big O notation. The algorithmic complexity of each code block would be dependent on the data structures used to implement the solutions. For instance, the code block $j \notin B$'s efficiency is $O(1)$ because the intents were stored in a bitwise array. If an integer array was chosen as the preferred data structure, then a binary search algorithm with an efficiency of $O(log\ |M|)$ would have to be used.

In the analysis of the specific building blocks that make up the eight different algorithms it can be seen clearly from in *Table 4.13* that the code blocks 3,7,8 and 16 have the highest complexity of $O(|M||G|)$. These code blocks correspond to the computation of full closure and partial closures. For the analytical comparison in addition to the full and partial closures we will also consider the complexity of computing the extents as it plays a major part in the algorithm.

Table 4.13, Algorithmic Complexity of Code Blocks

No	Code Blocks	Algorithmic Complexity of Code Block
1	$j \notin B$	$O(1)$
2	$C \leftarrow A \cup \{j\}^{\downarrow}$	$O(\|G\|)$
3	$D \leftarrow C^{\uparrow}$	$O(\|M\|\|G\|)$
4	$B \cap Y_j = D \cap Y_j$	$O(\|M\|)$
5	$A = C$	$O(1)$
6	$B \leftarrow B \cup \{j\}$	$O(1)$
7	$B = C^{\uparrow Y_j}$	$O(\|M\|\|G\|)$
8	$B \cap Y_j = C^{\uparrow Y_j}$	$O(\|M\|\|G\|)$
9	$PutInQueue(C, j)$	$O(1)$
10	$GetFromQueue(C, j)$	$O(1)$
11	$M^j \leftarrow N^j$	$O(1)$
12	$N^j \cap Y_j \subset B \cap Y_j$	$O(1)$
13	$M^j \leftarrow D$	$O(1)$
14	$PutInQueue((C, D), j)$	$O(1)$
15	$GetFromQueue((C, D), j)$	$O(1)$
16	$M^j \leftarrow C^{\uparrow Y_j}$	$O(\|M\|\|G\|)$
17	$D \leftarrow B \cup \{j\}$	$O(1)$

4.5.2 Analysis of CbO-FC-DF (CbO)

Let $g = |G|, m = |M|, c = |L|$

Let $a = \frac{1}{c}\sum_{r=1}^{c} |A_r|$

Always $a < g$

Let q be the number of times the statements inside the main if statement runs (line 4,5,6) (See *Figure 4.1*). Due to the similarity of the algorithms this number is a constant across the five variations of CbO.

Cost of the computation of intents (Line 5) = $q.m.a$(1)

Cost of the computation of extents (Line 4) = $q.a$ (2)

4.5.3 Analysis of CbO-PC-DF (In-Close)

CbO-PC-DF (In-Close) (See *Figure 4.3*) uses partial closure in line 6 as part of its canonicity test. The intents are actually calculated by using incremental closures with each operation taking the complexity of $O(1)$. In the CbO-FC-DF (CbO) algorithm, a concept is fully closed before the canonicity test. It is however sufficient to close the concept up to the current element j for the canonicity test (Andrews, 2014). There is a computational saving of $a(n-j)$ attributes for each iteration.

The definition of the partial closure operator is given below.

$$A^{\uparrow Z} := \{y \in Z \mid \forall x \in A : x\, I\, y\}$$

Let $y = \frac{1}{q}\sum_{r=1}^{q}(m - j_r)$

Here j_r corresponds the value of j in the inner loop for each time the partial closure is computed in line 6. y is the average number of attribute comparisons reduced in computing the partial attribute closure.

Always $0 < y < m$

Cost of the computations of intents (Line 6,4,7)

$$= q.(m-y).a + q \approx q.(m-y).a \qquad (3)$$

Cost of the computations of extents (Line 2)

$$= q'.a \approx q.a \qquad (4)$$

Here q' is the number of times the for loop runs. $q' > q$ (as q is the number of times the statements inside the if statement executes). Because $q.(m-y).a < q.m.a$ *(See (1) and (3))* we can conclude that the computational cost of CbO-PC-DF (In-Close) is lower than CbO-FC-DF (CbO).

4.5.4 Analysis of CbO-PC-DBF (In-Close2)

A queue is used in CbO-PC-DBF (In-Close2) (See *Figure 4.10*) for a combined breadth first and depth first approach. Due to the delayed recursion calls the intent calculation is complete before the recursive call. This is known as the attribute inheritance where a child element will get the parents attributes (Andrews, 2015).

Let s be the total saving in the attribute calculations in the children nodes for the entire set of concepts.

Cost of the computations of intents (Line 5,11,7)

$$= q.(m - y).a + q - s \approx q.(m - y).a \qquad (5)$$

Cost of the computations of extents (Line 3)

$$= q.a \qquad (6)$$

Thus we can conclude that the computational cost of CbO-PC-DBF (In-Close2) is lower than that of CbO-PC-DF (In-Close). Since $q.(m - y).a + q - s < q.(m - y).a + q$ *(See (3) and (5))* and $q.a < q'.a$ *(See (4) and (6))*

4.5.5 Analysis of CbO-FC-ICF-DBF (FCbO)

CbO-FC-ICF-DBF (FCbO) (See *Figure 4.12*) introduces inherited canonicity failures. The cannoncity test shown in line 7, fails for an attribute $j \notin B$, then the test will also fail for each $B' \supseteq B$ where $j \notin B'$, as long as $((D\setminus B) \cap Yj)$ contains an attribute which is not in B'. This information can be passed from one level to the next by recording the intent D of a failed canonicity test for an attribute j. This is stored in Mj in line 10 and in turn represented in Nj when the recursive call is made. This reduces the execution of the inner loop.

Let f be the total number of successful inherited canonicity failures(Andrews, 2015).

CbO-FC-ICF-DBF (FCbO) also uses a combined breadth first and depth first approach in recursion by using a queue.

Cost of the computations of intents (Line 6)

$$= (q - f).m.a - s \approx (q - f).m.a \qquad (7)$$

Cost of the computations of extents (Line 5)

$$= (q - f).a \qquad (8)$$

Thus we can conclude that the computational cost of CbO-FC-ICF-DBF (FCbO) is faster than CbO-FC-DF (CbO). Since $(q - f).m.a < q.m.a$ *(See (1) and (7))* and $(q - f).a < q.a$ *(See (2) and (8))*.

4.5.6 Analysis of CbO-FC-DBF

Thus the cost of the computations of intents and extents in CbO-FC-DBF (See *Figure 4.6*) is given below.

Cost of the computations of intents (Line 5) $= q.m.a - s$ (9)

Cost of the computations of extents (Line 4) $= q.a$ (10)

Because $q.m.a - s < q.m.a$ *(See (9) and (1))* we can conclude that CbO-FC-DBF has a lower computational cost than CbO-FC-DF (CbO).

4.5.7 Analysis of CbO-FC-ICF-DF

Thus the cost of the computations of intents and extents in CbO-FC-ICF-DF (See *Figure 4.8*) is given below.

Cost of the computations of intents (Line 6)

$$= (q - f).m.a \qquad (11)$$

Cost of the computations of extents (Line 5)

$$= (q - f).a \qquad (12)$$

Because $(q - f).m.a - s < (q - f).m.a$ *(See (9) and (11)).* We can conclude that CbO-FC-ICF-DBF (FCbO) has a lower computational cost than CbO-FC-ICF-DF

4.5.8 Analysis of CbO-PC-ICF-DF

Thus the cost of the computations of intents and extents in CbO-PC-ICF-DF (See *Figure 4.11*) is given below.

Cost of the computations of intents (Line 8,6,9)

$$= (q - f).(m - y).a + (q - f) \approx (q - f).(m - y).a \qquad (13)$$

Cost of the computations of extents (Line 5)

$$= (q - f).a \qquad (14)$$

CbO-PC-ICF-DF has a lower computational cost compared to CbO-FC-ICF-DF because $(q - f).(m - y).a < (q - f).m.a$ *(See (13) and (11))*

4.5.9 Analysis of CbO-PC-ICF-DBF (In-Close3)

In-Close3 also has inherited canonicity failures. The major difference between CbO-FC-ICF-DBF (FCbO) and CbO-PC-ICF-DBF (In-Close3) is that the latter uses partial closures. Thus the cost of the computations of intents and extents in CbO-PC-ICF-DBF (In-Close3) (See *Figure 4.13*) is as follows.

Computations of intents (Line 8,6,14)

$$= (q-f).(m-y).a \,+\, (q-f) - s \approx (q-f).(m-y).a \qquad (15)$$

Computations of extents (Line 4)

$$= (q-f).a \qquad (16)$$

Because $(q-f).(m-y).a \,<\, q.(m-y).a$ *(See (5) and (15))* and $(q-f).a \,<\, q.a$ *(See (6) and (16))*. We can conclude that CbO-PC-ICF-DBF (In-Close3) has a lower computational cost than CbO-PC-DBF (In-Close2). We can also conclude that CbO-PC-ICF-DBF (In-Close3) has a lower computational cost than CbO-FC-ICF-DBF (FCbO) Since $(q-f).(m-y).a \,<\, (n-f).m.a$ *(See (7) and (15))*. We can assume that the value of $C^{\uparrow Yj}$ which is needed for line 11 is cached when it is originally computed for line 8.CbO-PC-ICF-DBF (In-Close3) has a lower computational cost than CbO-PC-ICF-DF because $(q-f).(m-y).a \,+\, (q-f) - s \,<\, (q-f).(m-y).a + (q-f)$ *(See (15) and (13))*. Finally CbO-PC-ICF-DBF (In-Close3) has a lower computation cost than CbO-FC-DBF since $(q-f).(m-y).a \,+\, (q-f) - s \,<\, q.m.a - s$ *(See (15) and (9))*.

Table 4.14, Analytical Results obtained

No	Analytical Results presented
1	CbO-PC-DF (In-Close) < CbO-FC-DF (CbO)
2	CbO-PC-DBF (In-Close2) < CbO-PC-DF (In-Close)
3	CbO-FC-ICF-DBF (FCbO) < CbO-FC-DF (CbO).
4	CbO-FC-DBF < CbO-FC-DF (CbO)
5	CbO-FC-ICF-DBF (FCbO) < CbO-FC-ICF-DF
6	CbO-PC-ICF-DF < CbO-FC-ICF-DF
7	CbO-PC-ICF-DBF (In-Close3) < CbO-PC-DBF (In-Close2)
8	CbO-PC-ICF-DBF (In-Close3) < CbO-FC-ICF-DBF (FCbO)
9	CbO-PC-ICF-DBF (In-Close3) < CbO-PC-ICF-DF
10	CbO-PC-ICF-DBF (In-Close3) < CbO-FC-DBF

The analytical results obtained in the paper are summarized in *Table 4.14*. These can be combined to produce the following results.

CbO-PC-ICF-DBF (In-Close3) < CbO-PC-DBF (In-Close2) < CbO-PC-DF (In-Close) < CbO-FC-DF (CbO)

CbO-PC-ICF-DBF (In-Close3) < CbO-FC-ICF-DBF (FCbO) < CbO-FC-DF (CbO).

CbO-PC-ICF-DBF (In-Close3) < CbO-PC-ICF-DF < CbO-FC-ICF-DF

CbO-FC-ICF-DBF (FCbO) < CbO-FC-ICF-DF

CbO-FC-DBF < CbO-FC-DF (CbO)

Eight of the ten analytical results presented in hold true for both the real world datasets and the random artificial datasets presented in Section 4.4. The second and fourth analytical results CbO-PC-DBF (In-Close2) < CbO-PC-DF (In-Close) and CbO-FC-DBF < CbO-FC-DF (CbO) both don't hold for the artificial random dataset results presented in Tables. However both hold true for the random datasets where the attribute size is 1000 and above (See *Table 4.7*). The main difference the algorithms in questions is depth first search vs a combined depth first and breadth first search feature used in isolation. The two results hold true for the Mushroom, Internet Ads and Adult, Internet Ads real world datasets respectively. Interestingly Internet Ads is the only real world dataset with over 1,000 attributes (See *Table 4.4*). This suggests that the combined depth first and breadth first feature requires larger attributes to make an impact.

5 – SHARED MEMORY PARALLEL ALGORITHMS

5.1 Different types of Parallel Machines

All computing devices used today are parallel machines. The introduction of multicore processors commenced around the year 2004 to solve the so called power wall problem. Prior to this CPU manufacturers resorted to increase the clock speed of each new generation of CPU eventually reaching the critical power consumption of 130 Watts around 2004. Beyond this point it was not economically possible to dissipate the heat produced by the CPU's. Over the last decade CPU manufacturers have kept the clock speed and core size of a CPU as constants and have resorted instead to add extra cores to a single die in the CPU to get better performance (Chappell & Stokes, 2012)

Today's laptops, desktop machines have at least two to four cores in the CPU. High end Xeon Processors have up to 24 cores. The latest high end Xeon Phi processors have up to 72 cores, where each core has the power of a single Intel Atom processor (Jeffers, Reinders, & Sodani, 2016).

Computer programs must be designed and implemented using a parallel approach to leverage on the multiple cores available in the CPU (Sutter & Larus, 2005). Traditional serial programs can only make use of one CPU core of the computer.

Today's computers are essentially Shared Memory Multiple Instructions, Multiple Data (MIMD) machines. They typically also support vector operations which are Single Instruction, Multiple Data (SIMD). The shared memory model simplifies the transactions between the CPUs. However this also constitutes a bottleneck and limits the scalability of the system (Barlas, 2014). Intel's new highly parallel many core CPU the Xeon Phi processor family have up to 72 cores running up to 288 threads with 512 bit vector instructions (Jeffers et al., 2016).

Distributed memory Multiple Instructions, Multiple Data (MIMD) machines are the other type of parallel machines that are available. These machines are made up of processors that communicate by exchanging messages. The communication cost is high, but since memory is not shared, such machines can scale well. Clusters and Supercomputers are examples of such machines (Barlas, 2014).

5.2 Developing Parallel Programs

Due to the nature on how shared memory and distributed memory parallel machines work there are two fundamentally different programming approaches that are used.

Shared memory parallel machines can be programmed by generating threads, where each thread is executed on a separate core. Shared memory programming models such as ArBB, TBB, OpenMP, Intel Cilk Plus have a built in scheduler which spawns the parallel processes based on the intent of parallelization given by the developer. The language constructs allow the developer to specify that a repetition structure such as the for loop should execute in parallel. Using this approach for instance a matrix manipulation which uses a for loop can be distributed across multiple core, where each core executes part of the computation. The code needs to be written in such a way that within the for loop there are no dependencies between the iterations. Another language construct allows the developer to give the intent of spawning the execution of a function on a separate core. The actual execution of this spawn function is carried out by the built-in scheduler, when resources are available. A common cause of errors in shared memory programming is sharing variables across cores. Typically, when a shared memory variable is updated only one of the cores should have the privilege to do so. However since frequent locking of shared memory variables degrades overall performance, developers make use of programming techniques where access to shared memory variables in code is minimized (Diaz, Munoz-Caro, & Nino, 2012; Jeffers & Reinders, 2015; Jeffers et al., 2016).

Another common type of shared memory programming technique is vectorization. Modern CPUs support data parallel instructions where one vector instruction can be executed on multiple data items. For instance, one vector addition instruction can be executed involving multiple numbers simultaneously. A typical Intel Xeon processor could execute eight 32 bit integer operations simultaneously using vector instructions. General Purpose Graphics Processing Units (GP-GPU) processors produced by Nvidia on the other hand are exclusively vector processors where thousands of integers can be used in a single vector instruction simultaneously (Alessandrini, 2015; Yang, Huang, & Lin, 2011).

Distributed Memory parallel computers do not share memory, so the only communication mechanism is to pass messages to invoke parallel processes and to send and receive data. Message Passing programming frameworks is the predominant programming model in distributed memory parallel computers. These computers are referred to as clusters. A cluster is made up of multiple stand-alone computers connected through a fast network and typically each computer that is used in the execution of the parallel program in the cluster is loaded with the same program. The term node is used to denote each computer in the cluster. Each node has a unique identification and it is used to group computers into workers and masters. In a typical arrangement the master node would generate parallel computational tasks which are sent through message passing to the worker nodes. The workers would perform the computation and the send the results back to the master nodes. MPI is the standard programming framework used in programming distributed parallel computers (Pacheco, 1997; Prasad, Gupta, Rosenberg, Sussman, & Weems, 2015; Snir, Otto, Huss-Lederman, Walker, & Dongarra, 1998).

It is also possible to develop hybrid parallel solutions; increasingly even super computers have nodes which consist of general CPUs and GP-GPUs. A parallel program could for instance be written using MPI, OpenMP and CUDA to leverage the capabilities of the nodes attached to a cluster (Kedia, 2009; Yang et al., 2011).

The parallel algorithms presented in this chapter are shared memory parallel algorithm.

5.3 Selection of Technology Stack for Shared Memory Parallel Implementation

The OpenMP command task was used to spawn new threads. High level shared memory thread programming frameworks such as OpenMP, Intel Cilk Plus have built in schedulers that are used to spawn threads. Developers only implicitly specify the intention of parallelization using appropriate commands in the code (Chappell & Stokes, 2012). A sophisticated runtime scheduler in the background handles the creation, assigning work and deletion of threads.

OpenMP is an industry standard shared memory programming framework which is supported by all the major C++ compilers. OpenMP is used widely as a shared memory parallel programming environment. On the other hand Intel Cilk Plus is proprietary, although some of its features have crept into the GNU compilers, it has now been officially depreciated from the Intel 2018 C++ compiler (Anoop, 2017).

The developer starts with a serial implementation of a code base and gradually converts it to a parallel program by inserting OpenMP directives which produces multithreaded code. This is one of the reasons that OpenMP is still popular. OpenMP implements a fork-join execution model. When a single running thread comes across a parallel directive, it activates a group of threads to execute multiple work. The OpenMP task command introduced in OpenMP 3 departs from the normal execution of having one task per thread allowing a mechanism for tasks to be queued to be executed later (Alessandrini, 2015).

5.4 Introduction on the Shared Memory Algorithms introduced

This chapter introduces three new shared memory algorithms. The selected serial algorithm for parallelization is the recursive In-Close3 algorithm. Since the concepts generated in each recursion sub tree is independent a naïve parallelization approach would be to spawn each recursive call as a thread running on a separate core. The Naïve Parallel In-Close3 algorithm is presented in *Figure 5.1*. This algorithm doesn't scale well.

The next two algorithms presented Direct Parallel In-Close3 and Queue Parallel In-Close3 gets its inspiration from the PFCbO algorithm. (Krajca, Outrata, & Vychodil, 2010a). In PFCbO the recursion level is used as a cutting point to spawn new threads. A predefined level is considered as a design optimization and each time the recursion level reaches this predefined level, a new thread would be spawned to compute all the concepts in that specific recursive sub tree. Krajca used a queue to store all the possible recursive sub trees that required computation. This was computed serially and once it was completed, the computation of concepts for each recursive sub trees were delegated to the different cores of the parallel machine. A similar approach was used in the Queue Parallel In-Close3 algorithm which is presented in *Figure 5.9*. This is the function executed in by the master or the main process. Once all the recursive sub tree calls are recorded in the queue, they are delegated to the different cores of the computer. Each core would run the original In-Close3 serial algorithm presented in *Figure 5.6*. The Direct Parallel In-Close3 algorithm presented in *Figure 5.8* doesn't require a queue, it directly spawns a thread to compute the concepts that are there in a given recursive sub tree. Here too the original serial In-Close3 algorithm given in *Figure 5.6* is used. These threads are executed on the different cores of the Shared Memory Computer.

The three new shared memory algorithms are presented in Section 5.5 and Section 5.6

5.5 Naive Parallel Algorithm for In-Close3

5.5.1 Introduction

The algorithm described in Algorithm 5.1 (See *Figure 5.1*) is the same serial In-Close3 algorithm described in Chapter 4 (See *Figure 4.13*). The only difference is in line 16 where the recursive call to the In-Close3 algorithm is invoked in parallel.

```
Algorithm 5.1: NAIVE PARALLEL IN-CLOSE3
ParallelComputeConceptsFrom $((A,B), y, \{N^y \mid y \in Y\})$
    for $j \leftarrow y$ upto $n-1$ do
        $M^j \leftarrow N^j$
        if $j \in B$ and $N^j \cup Y_j \subseteq B \cap Y_j$ then
            $C \leftarrow A \cap \{j\}^{\downarrow}$
            if $A = C$ then
                $B \leftarrow B \cup \{j\}$
            else
                if $B \cap Y_j = C^{\uparrow Y_j}$ then
                    PutInQueue$(C, j)$
                else
                    $M^j \leftarrow C^{\uparrow Y_j}$
    ProcessConcept$((A,B))$
    while GetFromQueue$(C, j)$ do
        $D \leftarrow B \cup \{j\}$
        spawn ParallelComputeConceptsFrom$((C,D),$
                                          $j+1, \{M^y \mid y \in Y\})$
```

Figure 5.1, Algorithm 5.1 : Naïve Parallel In-Close3

Here principally the algorithm spawns each recursive call as a separate thread running on a separate core. The word spawning is used in parallel computing to denote invocation of a new child process concurrently. High level shared thread programming frameworks such as OpenMP, Intel Cilk Plus support task parallelism where you can specify that a certain part of code needs to be executed in parallel. In OpenMP this is achieved through the omp task command, which however does not guaranty the execution order of the tasks. The Intel Cilk Plus shared memory library which uses the spawn command to achieve a similar effect using a different scheduling algorithm guaranties the order in which the given tasks are executed (Vladimirov, Asai, & Karpusenko, 2015).

5.5.2 Performance Evaluation of Naïve Parallel In-Close3

Table 5.1 shows the performance of the Naïve Parallel In-Close3 algorithm on a Colfax cluster node that used an Intel® Xeon ® Gold 6128 @ 3.7 GHz, six core processor with 96GB RAM, running a stripped down version of Suse Linux. This configuration is further described in the methodology Section 2.7.

Table 5.1, Average Time obtained for different real datasets, with 95% confidence levels

Data Set	Mushroom	Adult	Internet Ads	Student
|X| x |Y|	8,124 x 125	32,561 x 99	3279 x 15652	587 x 145
Density	17.36%	11.29%	0.97%	24.50%
# Concepts	226,920	80,332	16,570	22,760,242
Time (seconds)	0.518 ± 0.0068	0.182 ± 0.0049	0.107 ± 0.0042	44.470 ± 0.6656

Table 5.2, Average Time obtained for different large artificial datasets, with 95% confidence levels

Data Set	M710G120K	M10X30G120K	T1014D100K
|X| x |Y|	120,000 x 70	120,000 x 300	100,000 x1,000
Density	10.00%	03.33%	01.01%
# Concepts	1,166,343	4,570,498	2,347,376
Time (seconds)	2.734 ± 0.1127	18.442 ± 0.1101	12.818 ± 0.1297

The graphs in *Figure 5.2* and *Figure 5.3*, shows that there is no scalability in the Naïve Parallel In-Close3 solution. Speedup is defined as the time taken to run the parallel application using one process divided by the time taken to run the parallel process using n number of cores. The poor performance of the naïve algorithm implementation can be explained by the fact that newly spawned process only produces one concept. Another process is spawned to generate the next concept. The overhead of spawning a new thread frequently is significant compared to the gains in running the concept generation in parallel.

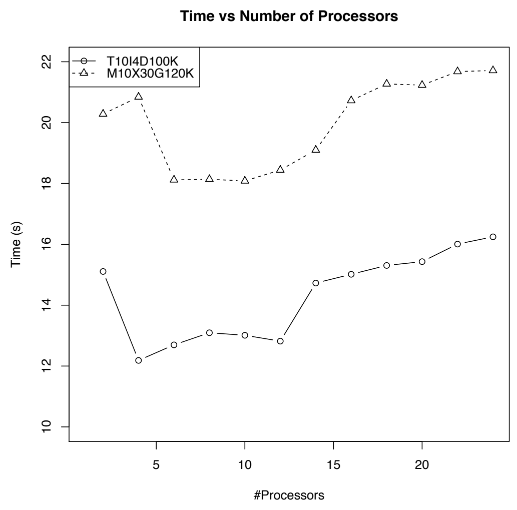


Figure 5.2, Naïve Parallel In-Close3, Timing

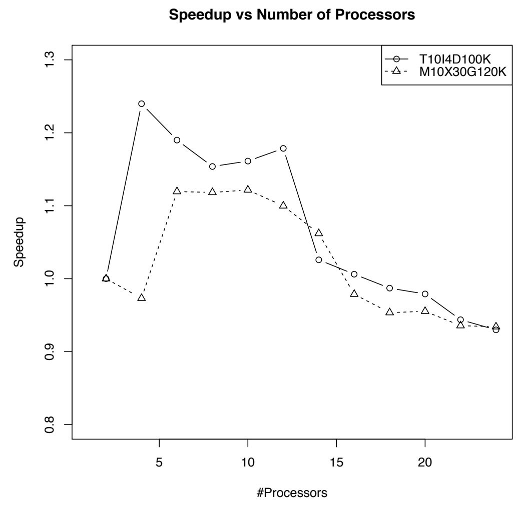


Figure 5.3, Naïve Parallel In-Close3, Speedup

5.6 Existing Parallel FCA Algorithms

Huaiguo Fu had created a parallel implementation of the NextClosure algorithm but it was limited to 50 attributes (Fu & Nguifo, 2004) but this was subsequently greatly extended (Fu & Foghlu, 2008). Krajca (Krajca, Outrata, & Vychodil, 2008) presented a parallel algorithm called PFCbO which parallelizes the FCbO algorithm. This is also a variation of the CbO algorithm(Kuznetsov & Obiedkov, 2002). Andrews's best of breeds In-Close3 is an improvement over the serial FCbO algorithm, where the key difference is the use of partial closures instead of full closures (Andrews, 2015). Krajca had used a queue specific to each thread to capture parameters of recursive sub call trees of a specific level of recursion (Krajca et al., 2008). Once all the sub call trees are captured, instances of threads are spawned in a round robin fashion to compute the remaining concepts in parallel.

```
Procedure GenerateFrom(⟨A, B⟩, y)
1  process $B$ (e.g., print $B$ on screen);
2  if $B = Y$ or $y > n$ then
3      return
4  end
5  for $j$ from $y$ upto $n$ do
6      if $B[j] = 0$ then
7          set $\langle C, D\rangle$ to ComputeClosure($\langle A, B\rangle, j$);
8          set $skip$ to false;
9          for $k$ from $0$ upto $j - 1$ do
10             if $D[k] \neq B[k]$ then
11                 set $skip$ to true;
12                 break for loop;
13             end
14         end
15         if $skip$ = false then
16             GenerateFrom($\langle C, D\rangle, j + 1$);
17         end
18     end
19 end
20 return
```

Figure 5.4, Algorithm 5.2 : GenerateFrom – Parallel FCbO (Krajca)

Figure 5.4 Algorithm 5.2 is essentially the serial version of FCbO as described by Krajca (Krajca et al., 2008). A general parallelization strategy described in (Keutzer & Mattson) is the Master Worker pattern where a master process generates a set of tasks which are later executed by delegating them to worker processes. Krajca, Outrata, & Vychodil, 2010 presented the PFCbO developed based on this pattern. The Master process `ParallelGenerateFrom()` initially generates a set of tasks by storing them in a queue which is shown in line 2 and 3 (See *Figure 5.5*). Once all the tasks are generated the algorithm proceeds to spawn workers by taking one item at a time from the queue. This is shown in lines 26 to 36 in *Figure 5.5*. The tasks are broken by a runtime optimization parameter L which represents the level of the recursion call sub tree. Each worker process computes all the concepts at that given level of the recursion subtree by calling `GenerateFrom()` (See *Figure 5.4*) in lines 29 and 34.

```
Procedure ParallelGenerateFrom($\langle A, B\rangle, y, l$)
if $l = L$ then
    select $r$ from 0 to $P-1$ (e.g. $r = (\sum_{s=0}^{P-1} queue[s]) \mod P$);
    store $(\langle A, B\rangle, y)$ to $queue[r]$;
    return
end
process $B$ (e.g., print $B$ on screen);
if $B = Y$ or $y > n$ then
    goto line 25;
end
for $j$ from $y$ upto $n$ do
    if $B[j] = 0$ then
        set $\langle C, D\rangle$ to ComputeClosure($\langle A, B\rangle, j$);
        set $skip$ to false;
        for $k$ from 0 upto $j-1$ do
            if $D[k] \neq B[k]$ then
                set $skip$ to true;
                break for loop;
            end
        end
        if $skip$ = false then
            ParallelGenerateFrom($\langle C, D\rangle, j+1, l+1$);
        end
    end
end
if $l = 0$ then
    for $r$ from 1 upto $P-1$ do
        new process
            while set $(\langle C, D\rangle, j)$ to load from $queue[r]$ do
                GenerateFrom($\langle C, D\rangle, j$);
            end
        end
    end
    while set $(\langle C, D\rangle, j)$ to load from $queue[0]$ do
        GenerateFrom($\langle C, D\rangle, j$);
    end
end
return
```

Figure 5.5, Algorithm 5.3 : ParallelGenerateFrom – Parallel FCbO (Krajca)

5.7 Two additional Parallel variations of In-Close3

Algorithm 5.4: COMPUTECONCEPTSFROM - PARALLEL-IN-CLOSE3

```
ComputeConceptsFrom $((A, B), y, \{N^y \mid y \in Y\}, level)$
    for $j \leftarrow y$ upto $n - 1$ do
        $M^j \leftarrow N^j$
        if $j \in B$ and $N^j \cup Y_j \subseteq B \cap Y_j$ then
            $C \leftarrow A \cap \{j\}^{\downarrow}$
            if $A = C$ then
                $B \leftarrow B \cup \{j\}$
            else
                if $B \cap Y_j = C^{\uparrow Y_j}$ then
                    PutInQueue$(C, j)$
                else
                    $M^j \leftarrow C^{\uparrow Y_j}$
    ProcessConcept$((A, B))$
    while GetFromQueue$(C, j)$ do
        $D \leftarrow B \cup \{j\}$
        ComputeConceptsFrom$((C, D), j + 1, \{M^y \mid y \in Y\}, level + 1)$
```

Figure 5.6, Algorithm 5.4 – ComputeConceptsFrom – Parallel-In-Close3

Two additional shared memory parallel In-Close3 algorithms are presented in this thesis. They are the Direct Parallel In-Close3 algorithm (See *Figure 5.8*) and the Queue Parallel In-Close3 (See *Figure 5.9*) algorithm. Both algorithms have got inspiration from Krajca's approach of parallelization where an entire recursion subtree is assigned to each thread. The Direct Parallel In-Close3 algorithm consists of two functions `Parallel_ComputeConceptsFrom()` (See *Figure 5.8*) and `ComputeConceptsFrom()` (See *Figure 5.6*). `ComputeConceptsFrom()` is essentially Andrews's serial version of In-Close3 (Andrews, 2015) with the exception of the additional parameter called level which keeps track of the current recursion level.

In addition the `Parallel_ComputeConceptsFrom()` function (See *Figure 5.8*) is identical to the `ComputeConceptsFrom()` function (See *Figure 5.6*) with the omission of the first two lines. The dashed lines shown in *Figure 5.8*, indicate the lines which are different from *Figure 5.6*). Similar to Krajca's approach the `Parallel_ComputeConceptsFrom()` function is executed serially by the main process. This algorithm calls the `ComputeConceptsFrom()` function in parallel (line 3 of *Figure 5.8*). The `Parallel_ComputeConceptsFrom()` function is invoked with $(A, B) = (X, X^{\uparrow})$. Where X represents a complete set of objects. Initial attribute $y = 0$ and a set of empty $Ns, \{Ny = \emptyset \mid y \in Y\}$ and $level = 0$. These values are the same as that is used for the serial algorithm presented in *Figure 4.13*. The parameter level is used to keep track of the level of recursion. The constant LEVEL is an optimization parameter that is used to determine the recursion level at which separate processes are spawned with the task of computing all the concepts in a given recursive subtree (See line 2 of *Figure 5.8*). For instance if the constant LEVEL is set to two for the recursive call tree given in *Figure 5.7*, the tasks that would be assigned to the parallel threads would be $\{5,\{6,\{8,9,10\}\},7,\{11,\{14\}\},12,\{13,\{15,16\}\},17,\{18,\{19,20\}\}$. The first available thread would be assigned the task of computing concepts of the recursive sub tree 5. However, since there are no children in this sub tree the thread would complete the task as soon as the concepts of 5 are computed. The next available thread in the meantime would have been assigned the task of computing the concepts of the recursive sub tree 6. During the computation the same thread is used to compute the concepts of 8,9 and 10 which are discovered and computed at runtime. It is clear in this example that the workloads given to each thread would be different. This is one of the disadvantages of this proposed solution. Another is the fact that concepts upto LEVEL two are computed serially. In the above example the concepts for 1,2,3 and 4 are computed serially. The same disadvantages are there in Krajca's parallel solution as well.

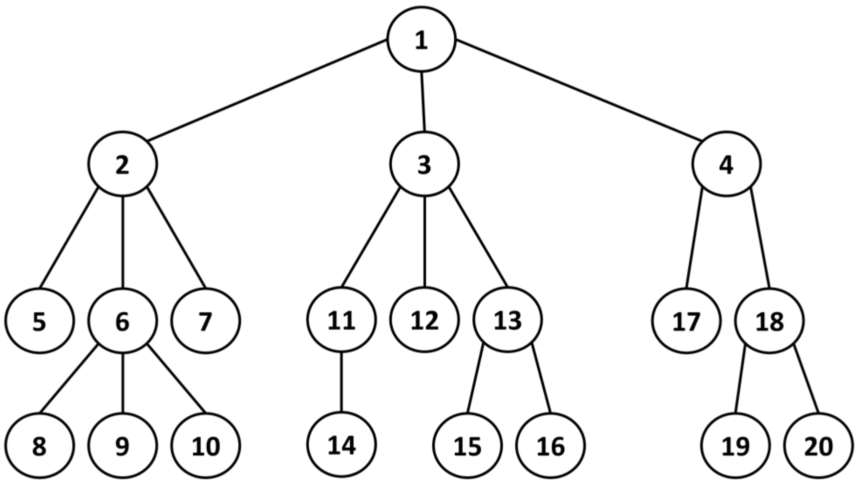


Figure 5.7, Combined Depth and Breadth First Recursive Call Tree

Krajca used separate queues to store each of the recursive call subtree workloads that were later distributed to separate threads in a round robin fashion (Krajca et al., 2008). The storing of tasks was done serially and the spawning of threads was carried out only after the computation of all the recursive call subtrees. The Direct Parallel In-Close3 algorithm spawns new threads as soon as they are discovered.

```
Algorithm 5.5: DIRECT PARALLEL-IN-CLOSE3
1  ParallelComputeConceptsFrom $((A,B), y, \{N^y \mid y \in Y\}, level)$
2      if $level = LEVEL$ then
3          spawn ComputeConceptsFrom$((C,D), j+1, \{M^y \mid y \in Y\}, level)$
4      for $j \leftarrow y$ upto $n-1$ do
5          $M^j \leftarrow N^j$
6          if $j \in B$ and $N^j \cup Y_j \subseteq B \cap Y_j$ then
7              $C \leftarrow A \cap \{j\}^{\downarrow}$
8              if $A = C$ then
9                  $B \leftarrow B \cup \{j\}$
10             else
11                 if $B \cap Y_j = C^{\uparrow Y_j}$ then
12                     PutInQueue$(C, j)$
13                 else
14                     $M^j \leftarrow C^{\uparrow Y_j}$
15     ProcessConcept$((A,B))$
16     while GetFromQueue$(C, j)$ do
17         $D \leftarrow B \cup \{j\}$
18         ParallelComputeConceptsFrom$((C,D), j+1, \{M^y \mid y \in Y\}, level+1)$
```

Figure 5.8, Algorithm 5.5 - Direct Parallel In-Close3 (the dash lines show the difference with Figure 5.6)

The second parallel In-Close3 algorithm proposed Queue Parallel In-Close3 (see *Figure 5.9*) is closer to Krajca's PCbO parallel algorithm where a queue is used to store tasks and are later assigned to workers. This algorithm also uses the same `ComputeConceptsFrom()` function (see *Figure 5.6*) for the worker processes to execute and to a lesser extent by the master process in computing concepts above the recursion call tree LEVEL. In essence in both Direct Parallel In-Close3 algorithm (See *Figure 5.8*) and the Queue Parallel In-Close3 algorithm (See *Figure 5.9*) the function `ParallelComputeConceptsFrom()` is executed serially by the master process while the `ComputeConceptFrom()` function is executed in parallel by worker processes.

Algorithm 5.6: QUEUE PARALLEL-IN-CLOSE3

```
1  ParallelComputeConceptsFrom $((A,B), y, \{N^y \mid y \in Y\}, level)$
2    if $level = LEVEL$ then
3      EnQueue$(A,B), y, \{N^y \mid y \in Y\}, level)$
4    for $j \leftarrow y$ upto $n-1$ do
5      $M^j \leftarrow N^j$
6      if $j \in B$ and $N^j \cup Y_j \subseteq B \cap Y_j$ then
7        $C \leftarrow A \cap \{j\}^{\downarrow}$
8        if $A = C$ then
9          $B \leftarrow B \cup \{j\}$
10       else
11         if $B \cap Y_j = C^{\uparrow Y_j}$ then
12           PutInQueue$(C, j)$
13         else
14           $M^j \leftarrow C^{\uparrow Y_j}$
15   ProcessConcept$((A,B))$
16   while GetFromQueue$(C, j)$ do
17     $D \leftarrow B \cup \{j\}$
18     ParallelComputeConceptsFrom$((C,D), j+1, \{M^y \mid y \in Y\}, level+1)$
19   if $level = 0$ then
20     while DeQueue$(C, D, j, \{M^y \mid y \in Y\}, level)$ do
21       spawn
           ComputeConceptsFrom$((C,D), j+1, \{M^y \mid y \in Y\}, level)$
```

Figure 5.9, Algorithm 5.6 - Queue Parallel In-Close3, (Dashed lines highlight the differences with Figure 5.8)

The only difference between the two algorithms is that in the Queue Parallel In-Close3 algorithm the tasks generated are executed only after all the tasks are computed. Whereas the Direct Parallel In-Close3 algorithm the tasks are ready to be executed as soon as they are generated, there is no separate explicit queue that is used for this purpose. In both OpenMP and Intel Cilk Plus the decision to spawn a thread is handled by the scheduling algorithm in the runtime environment of these parallel programming frameworks. Both frameworks can store a list of tasks that needs to be spawned.

The Queue Parallel In-Close3 algorithm was implemented using two strategies. The first approach called Simple Queue Parallel In-Close3 implementation uses an approach closer to Kraja's approach where in line 20 and 21 represent the queue of tasks (see *Figure 5.9).* The second variation called OpenMP Queue Parallel In-Close3 runs both line 20 and 21 as separate parallel threads and doesn't use the OpenMP task command.

5.8 Comparison of the four proposed parallel variations of In-Close3

The OpenMP implementation of the Parallel In-Close3 algorithms were executed on a single node of a Colfax cluster. By nature, a high end cluster provides dedicated access of the compute nodes required to run a program. This ensures that during the empirical testing that all the test results obtained are only as a result of the experimental program that was run. The values were computed with the LEVEL set as two. *Table 5.3* and *Table 5.4* and shows the average time obtained for each of the different real-world datasets and artificial datasets with LEVEL set as two and cores set to 12. The details of the compute node used in the experiments is the same that was used in Section 5.5.2. The 95% confidence level values are also shown in results. The Direct Parallel In-Close3 is the fasted algorithm based on the timings presented in *Table 5.3* and *Table 5.4*. The parallel FCbO (PFCbO) algorithm's parallelization strategy is similar to the strategy used by OpenMP Queue Parallel In-Close3. Hence to carry out a proper comparison the PFCbO algorithm was implemented similar to the OpenMP Queue Parallel In-Close3 coding. The comparison between these two algorithms clearly show that the proposed OpenMP Queue Parallel In-Close3 is faster for all datasets that were considered. A proper statistical analysis of the results presented in *Table 5.3* and *Table 5.4* are presented in Section 5.9.2

Table 5.3, Real World Dataset Results (average timing in seconds) for In-Close3 and FCbO parallel algorithms for LEVEL = 2 and Cores = 12 with 95% confidence level

Data Set	Mushroom	Adult	Internet Ads	Student
|G| x |M|	8,124x125	32,561x99	3279x1565	587 x 145
Density	17.36%	11.29%	0.97%	24.50%
# Concepts	226,920	80,332	16570	22,760,242
Naïve Parallel In-Close3	0.518 ± 0.0068	0.182 ± 0.0049	0.107 ± 0.0042	44.470 ± 0.6656
Direct Parallel In-Close3	0.105 ± 0.0049	0.045 ± 0.0024	0.024 ± 0.0011	11.325 ± 0.0226
Simple Queue Parallel In-Close3	0.115 ± 0.0041	0.052 ± 0.0018	0.063 ± 0.0020	11.229 ± 0.0523
OpenMP Queue Parallel In-Close3	0.114 ± 0.0033	0.054 ± 0.0025	0.064 ± 0.0032	10.776 ± 0.0426
OpenMP Queue Parallel FCbO	0.176 ± 0.0005	0.088 ± 0.0073	0.247 ± 0.0002	11.157 ± 0.1108

Table 5.4, Artificial Dataset Results (average timing in seconds) for In-Close3 and FCbO parallel algorithms for LEVEL = 2 and Cores = 12 with 95% confidence level

Data Set	M710G120K	M10X30G120K	T1014D100K
|G| x |M|	120,000 x 70	120,000 x 300	100,000 x1,000
Density	10.00%	03.33%	01.01%
# Concepts	1,166,343	4,570,498	2,347,376
Naïve Parallel In-Close3	2.734 ± 0.1127	18.442 ± 0.1101	12.818 ± 0.1297
Direct Parallel In-Close3	0.659 ± 0.0007	07.002 ± 0.0140	04.187 ± 0.0059
Simple Queue Parallel In-Close3	0.749 ± 0.0013	07.353 ± 0.0099	04.992 ± 0.0090
OpenMP Queue Parallel In-Close3	0.698 ± 0.0008	08.015 ± 0.1331	05.306 ± 0.1081
OpenMP Queue Parallel FCbO	0.966 ± 0.0018	25.580 ± 0.1635	16.554 ± 0.1542

The graphs shown in *Figure 5.10, Figure 5.11* and *Figure 5.12* shows the relative speedup of running the datasets M10X30G120K and T1014D100K for the Direct Parallel In-Close3, Simple Queue Parallel In-Close3 and the OpenMP Queue Parallel In-Close3 implementations. There is a clear drop in performance when then core count reaches four (See *Figure 5.10, Figure 5.11* and *Figure 5.12*). This can be seen across all datasets that were considered. This could be explained due to the fact that the Intel® Xeon ® Gold 6128 processor used in the experimental setup has only six physical cores.

Figure 5.13 shows how the implementation behaved for different values of LEVEL. The best results are obtained when LEVEL is set to one. Krajca had reported best results when LEVEL had the value of two (Krajca et al., 2008). Here too the behaviour of the Internet ad, M10X30G120K and T1014D100K dataset is slightly different where it picks up an improved performance for LEVEL = 4 and 5. *Table 5.5*, shows the number of threads called for different values of LEVEL for the Mushroom, Adult and Internet Ad datasets.

Table 5.5, No of threads called in parallel for different values of LEVEL

Level	**Mushroom**	**Adult**	**Ad**	**Example**
0	1	1	1	1
1	35	91	371	3
2	398	1,505	2,041	8
3	2,307	8,722	4,051	8
4	8,261	22,259	4,138	0
5	20,358	26,167	3,003	0

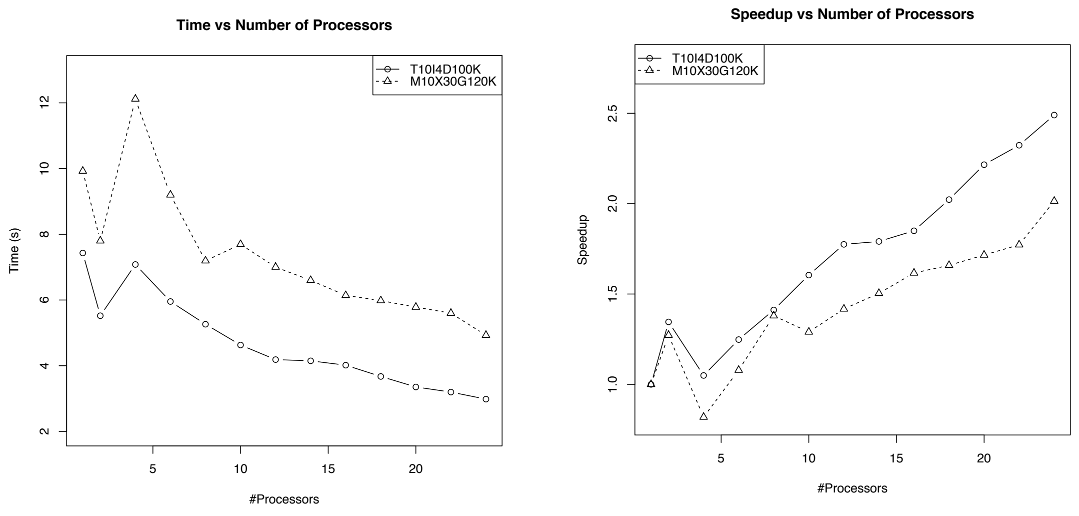


Figure 5.10, Direct Parallel In-Close3

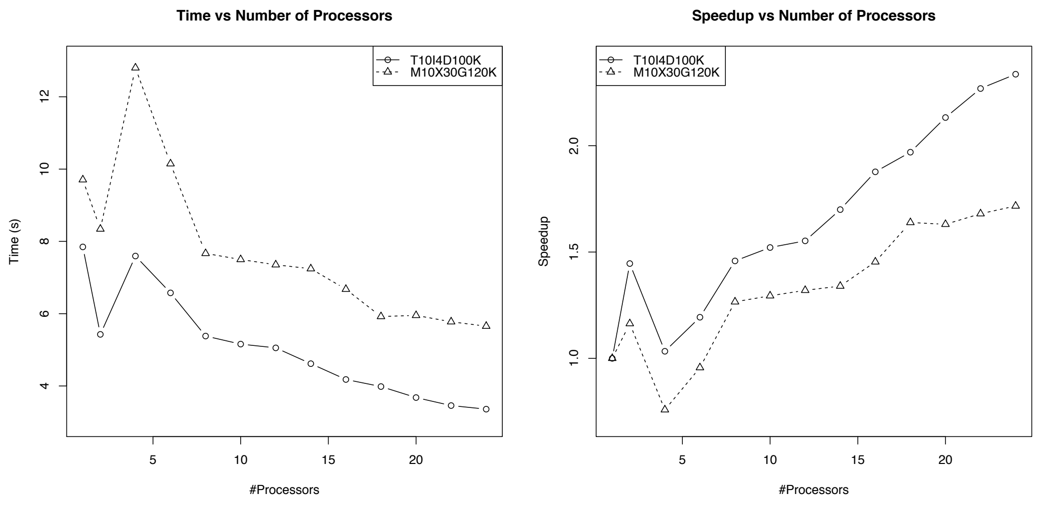
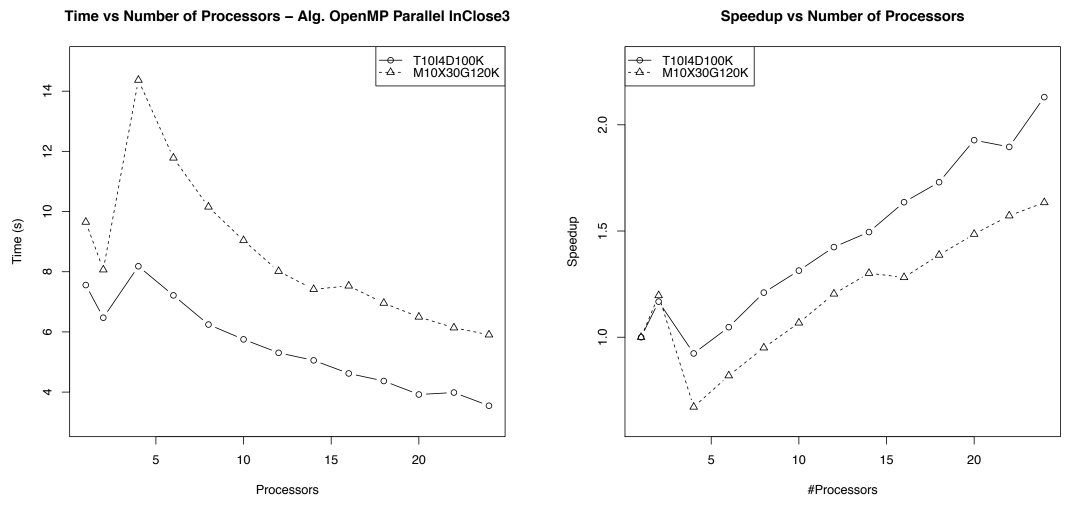


Figure 5.11, Simple Queue Parallel in-Close3



Figure

Figure 5.12, OpenMP Parallel In-Close3

For example for LEVEL = 1, the mushroom dataset produces 35 parallel threads. For LEVEL = 2 the number of parallel threads called increases to 398. The values for LEVEL = 0 and 1 are computed serially. Hence 36 recursive calls are handled serially. For LEVEL = 3 the number of parallel threads increases to 2307 and 398+35+1 calls are carried out serially. The reduction of performance as the LEVEL increases can be explained by the increase of the serial component of the computation and that the work assigned to each parallel process gets smaller as LEVEL increases. For instance, in LEVEL = 2, 398 processes were responsible for the computation of roughly 200,000 concepts, where as in LEVEL = 3, the amount of processes had increased to 2,307 to compute a lesser amount of concepts.

Figure 5.14, show details of an experiment of a series of artificial datasets where the density of the context was varied from 20 to 50 and the attributes and objects kept at 100 each. All the algorithms show an exponential growth as the density increases for LEVEL = 2 and the number of processes = 12.

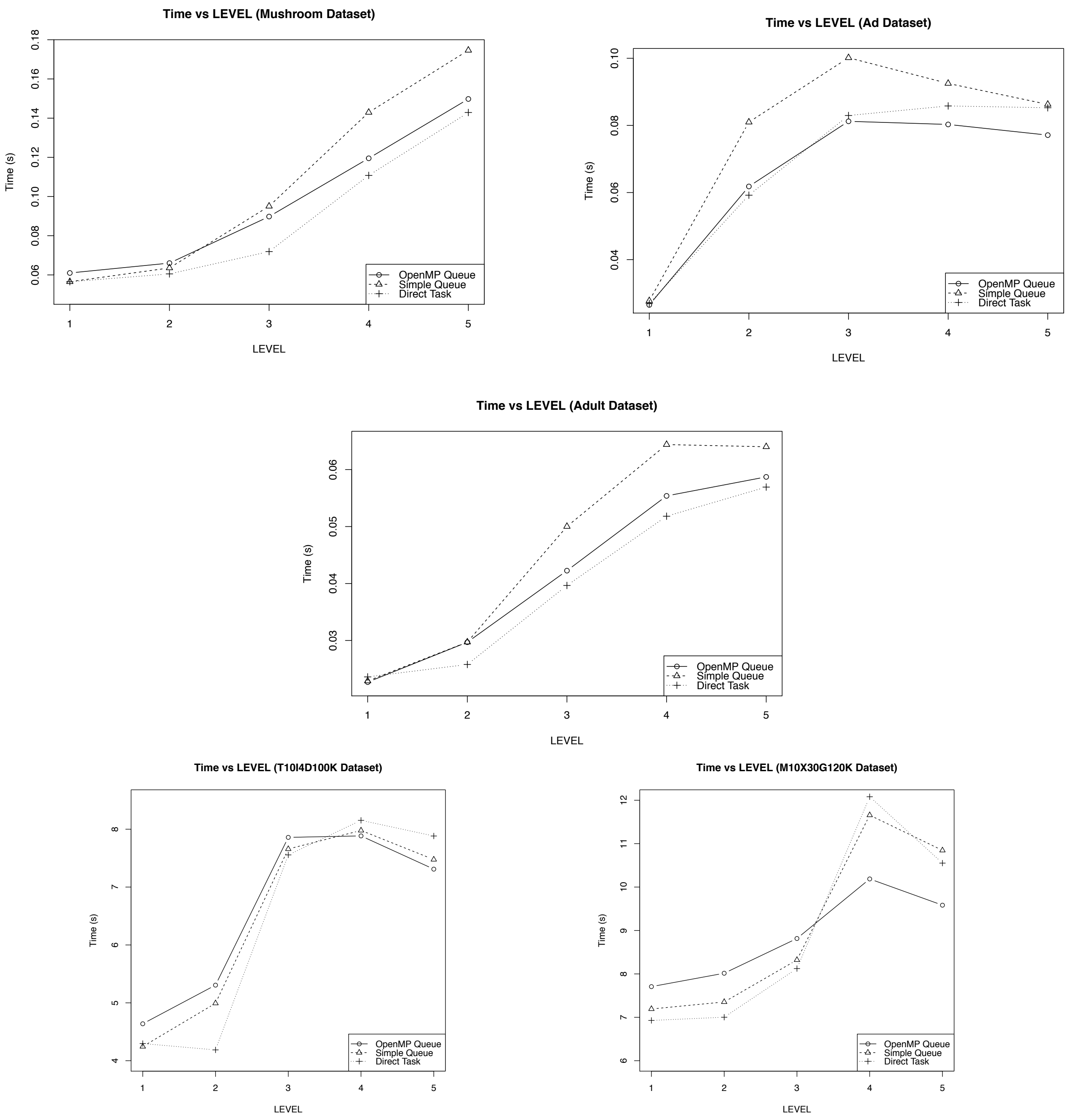


Figure 5.13, (a) Mushroom, (b) Ad, (c) Adult, (d) T1014D100K (e) M10X30G120K dataset vs Levels for Parallel In-Close3 Algorithms

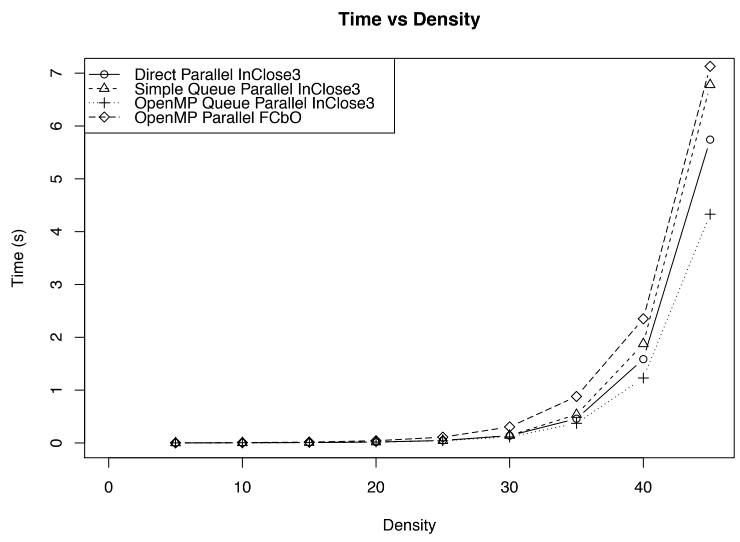


Figure 5.14, Time vs Density for LEVEL = 2, Cores - 12

Figure 5.15 show details of how the algorithms perform in an experiment where the objects of the context was varied from 10,000 to 100,000 for LEVEL = 2 and the number of processes = 12. The growth shows a linear relationship. Here the attributes were fixed at 100 and the density was fixed at 5%.

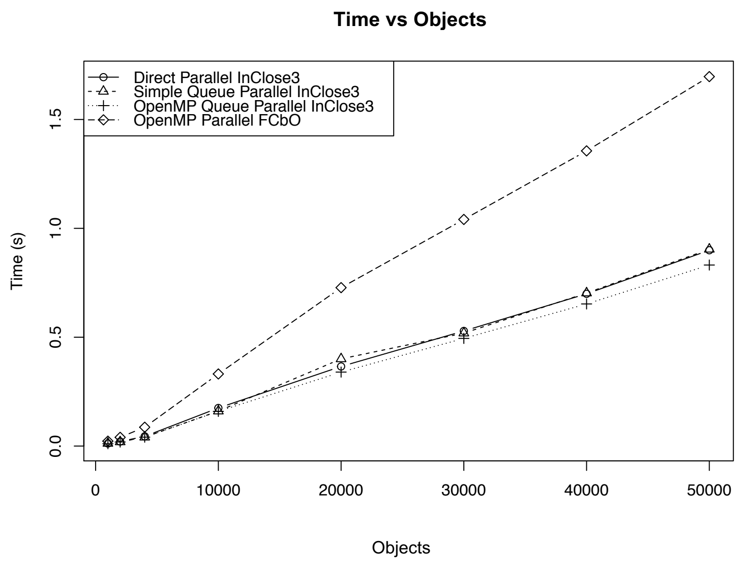


Figure 5.15 , Time vs Objects for LEVEL = 2, Cores - 12

Finally, *Figure 5.16* show the performance of the algorithms as attributes were varied from 1000 to 2000. The objects and density of the contexts were fixed to 100 and 5% respectively.

The OpenMP version of PFCbO performed poorly justifying the relatively poor results it obtained for the Internet Ads dataset in *Table 5.3.*

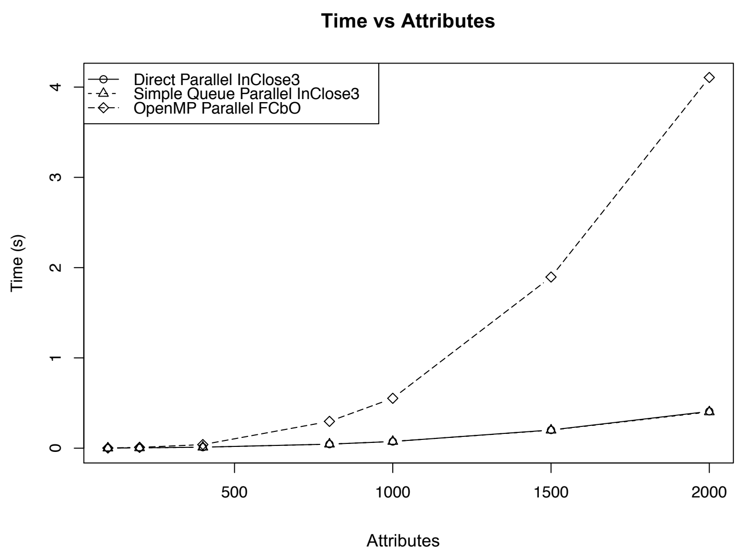


Figure 5.16, Attributes vs Time for LEVEL = 2, Cores - 12

5.9 Comparison with PFCbO

5.9.1 Introduction

To compare the parallel In-Close3 algorithm's performance with contemporary parallel CbO algorithms, a parallel version of FCbO (PFCbO) was also developed. Both implementations were derived from the serial codebase for In-Close3 and FCbO which was presented in Chapter 4. The data structures used were modified for the FCbO variation using exactly the same approach as of parallel In-Close3. Since the Parallel version of FCbO used a queue two implementations of the parallel algorithms were made. OpenMP Queue Parallel FCbO and Simple Queue Parallel FCbO are identical to their In-Close3 variants OpenMP Queue Parallel In-Close3 and Simple Queue Parallel In-Close3 respectively with the exception of the recursive algorithms `ComputeConceptsFrom()` and `ParallelComputeConceptsFrom()`. The PFCbO variants had their specific implementations of these recursive functions.

Table 5.3, Table 5.4, Figure 5.14, Figure 5.15 and Figure 5.16 show the comparison of the PFCbO algorithm implementations with the In-Close3. The variation is similar to the serial implementation comparison presented in Chapter 4. In-Close3 parallel variations performed better than parallel FCbO variations in both random and real world datasets. The PFCbO implementation performed comparatively better for the Mushroom and Adults datasets compared to the Internet Ad dataset (See *Table 5.3*). The graph shown in *Figure 5.15* highlighting the variation of attributes also shows that the PFCbO implementation performs poorly when attributes are varied.

5.9.2 Statistical Significance of the empirical results

The results obtained in *Table 5.3* and *Table 5.4* were checked to see if they had statistical significance by first carrying out one-way analysis of variance (one-way Anova).

The null hypothesis in one-way analysis H_0 is that the means of the empirical timing of all the seven algorithms for a given dataset are the same. We can clearly see that the conditions $F > F_{crit}$ and $P_{value} < \alpha$ is valid for all six datasets that were analyzed. Here α was taken as 0.05. *Table 5.6* shows a summary of the one-way analysis.

Table 5.6, One-way Analysis of the real-world datasets used in the analysis

dataset	F	F_{crit}	P_{value}
M10X30G120K	35,087.5	2.5787	5.66E-78
T10I4D100K	11,572.8	2.8663	9.21E-54
M7X10G120K	1,584.1	2.5787	9.01E-48
Student	12,443.2	2.5787	7.53E-68
Ad	7,879.5	2.8663	9.23E-51
Mushroom	12,239.1	2.5787	1.09E-67
Adult	987.9	2.5787	3.36E-43

Hence according to the Anova analysis we can reject H_0 the null hypothesis. We can conclude that for each real-world dataset there exists at least one algorithm that has a mean which is statistically different from others. Section 2.3.2 shows in detail how the one-way analysis was carried out.

To show that the means of the top two algorithms for each dataset has statistical significance the t-Test for two sample assuming unequal variances analysis was carried out. *Table 5.7*, shows a summary of the t-Test.

Table 5.7, t-Test for two sample assuming unequal variances analysis for the best two algorithms for each dataset

dataset	t_{stat}	mean1	mean2	$t_{Critical\ two\text{-}tail}$	$P_{two\text{-}tail}$	algo1	algo2
M10X30G120K	-49.96	7.0021	7.3531	2.1098	6.89E-20	*Parallel Direct*	*Parallel Simple*
T10I4D100K	-172.81	4.1875	4.9923	2.1314	3.65E-26	*Parallel Direct*	*Parallel Simple*
M7X10G120K	-97.08	0.6585	0.7005	2.1098	8.95E-25	*Parallel Direct*	*Parallel OpenMP*
Student	-8.21	10.7766	11.1604	2.1788	2.89E-06	*Parallel OpenMP*	*PFCbO*
Ad	-17.74	0.0476	0.0634	2.2622	2.61E-08	*Parallel Direct*	*Parallel Simple*
mushroom	-4.03	0.1041	0.1143	2.1199	9.59E-04	*Parallel Direct*	*Parallel OpenMP*
Adult	-5.47	0.0460	0.0516	2.1098	4.17E-05	*Parallel Direct*	*Parallel Simple*

The null hypothesis in the t-Test analysis H_0 is that the mean values of the two fastest algorithms considered are the same. For all the datasets in *Table 5.7* we can clearly see that $|t_{stat}| > t_{critical\ two-tail}$ and $P_{two-tail} < \alpha$ is valid. Here α was taken as 0.05. Here based on the t-Test we can reject H_0 the null hypothesis. Hence, we can conclude that Parallel Direct In-Close3 is the fastest algorithm for the all the datasets with the exception of the Student dataset where the OpenMP Queue Parallel In-Close3 is the fastest algorithm.

To show that the OpenMP Queue Parallel In-Close3 is faster than the OpenMP Queue Parallel implementation of PFCbO the t-Test analysis was carried out for those result sets.

Table 5.8, t-Test for two sample assuming unequal variances analysis for parallel In-Close3 and parallel FCbO

dataset	t_{stat}	mean1	mean2	$t_{Critical\ two\ tail}$	$P_{two\text{-}tail}$	algo1	algo2
M10X30G120K	-201.95	8.0154	25.5723	2.1009	1.17E-31	*In-Close3 Parallel*	*PFCbO*
T10I4D100K	-198.15	5.3060	16.5543	2.1314	4.69E-27	*In-Close3 Parallel*	*PFCbO*
M7X10G120K	-336.15	0.7005	0.9658	2.1604	5.40E-27	*In-Close3 Parallel*	*PFCbO*
student	-8.21	10.7766	11.1604	2.1788	2.89E-06	*In-Close3 Parallel*	*PFCbO*
Ad	-17.74	0.0476	0.0634	2.2622	2.61E-08	*In-Close3 Parallel*	*PFCbO*
mushroom	-41.91	0.1143	0.1764	2.2622	1.25E-11	*In-Close3 Parallel*	*PFCbO*
Adult	-10.59	0.0541	0.0888	2.2010	4.17E-07	*In-Close3 Parallel*	*PFCbO*

For all the datasets in *Table 5.8* we can clearly see that $|t_{stat}| > t_{critical\ two-tail}$ and $P_{two-tail} < 0.05$ is valid. Here we can reject H_0 the null hypothesis. We can conclude that OpenMP Queue Parallel In-Close3 is faster for the all the datasets than the OpenMP Queue PFCbO.

5.10 Implementation Details

For the serial implementation of the CbO algorithms predefined large one dimensional arrays were used to store the extents and intents of the concepts that were being incrementally generated. A structure was used to capture the other variables would be needed to support storing the extents and intents in a one dimensional array. These structures are referred to as scratchpads in this section. They effectively eliminated the need to use dynamic memory allocation to allocate memory on the fly whenever a new concept was computed.

```
typedef struct  {
        int *values;
        unsigned int *Start;
        unsigned int *End;
        unsigned int mstack; // pointer to last storage
        unsigned int maxIndex; // Maxindex
} TLa;


typedef struct  {
        short int *values;
        unsigned int *Start;
        unsigned int *End;
        unsigned int mstack; // pointer to last storage
        unsigned int maxIndex; // Maxindex
} TLb;
```

Figure 5.17 , Structure of Scratchpad used to store concepts generated

In *Figure 5.17* `TLa` is the scratchpad that captures the extents while `TLb` captures the intents. The `values`, `Start` and `End` are dynamically allocated arrays and are predefined during program initialization. The `values` property stores the actual data values. The `Start` and `End` store the starting and end point of each generated concept.

For instance, if we take a variable of the `TLa` datatype as defined in the main program `La`. A sample set of values for La is shown in *Figure 5.18.*

`La.values`

1	2	7	10	4	11	2	5	9	11	1	2	6	13	22		

`La.Start`

0	4	6	10		

`La.End`

3	5	9	14		

Figure 5.18 , Sample values of Scratchpad

The generated concepts in the serial implementation is stored in an integer arrays with the names `extents[]` and `intents[]`

For the above example sample values for `extents[0] = 0, extents[1] = 2, extents[2] = 3, extents[3] = 1`. These correspond to indices of La.Start and La.End.

For instance the second generated extent (extents[1] has the value 2 which correspond to `La.Start[2] = 6` and `La.End[2] = 9.` The extent is the values stored in indices `6,7,8,9` of the array `La.values {2,5,9,11}.`

A direct approach to do this would have been to use pointers for `extents[]` and `intents[]` and directly point to the specific values in `La.values`. However using pointers has potential drawback in vectorization and cannot be directly used in distributed memory solutions.

For parallel programs using a single common scratchpad such `La` for storing the extents being incrementally generated will create a bottleneck as concepts are generated in parallel. If such an approach is required a mutex would be required allowing only one process to access the `La.values` array, for storing extents.

A better strategy would be to have separate scratchpads and the approach used in parallelization is to have an array of scratchpads. For instance, `La` for the shared memory implementation is an array of the `TLa` structure having the array size equal to the number of processors that are used.

Algorithms that use partial closures incrementally build the intents and a B-Tree structure is used to capture the intent being generated for storage purposes. For computations a bit array version of the complete intent being generated is used. With the parallel implementation of the algorithms that use partial closures, parts of the intent can be stored in different processes. In building up the intents that are stored during concept generation, the process in which part of the intent was built needs to be captured.

For the naïve parallel implementation, each recursion call can run in any available processor. This requires the capture of the processor id of each part of the intent that is computed in the B-Tree structure.

In the other three implementation strategies, where a complete recursive sub tree is computed by one process guaranties that parts of the intent at most is computed by only two processes. The later part of the intent will be in the process where the concept is generated, and the initial part of the intent will be in the main process. It is also possible for an intent to be completely defined in the main process or be completely defined in a separate process. The modification needed for B-Tree structure to capture this is far simpler than the naïve parallel implementation.

The size of the scratchpad is a startup constant, before storing a new intent or extent the code would check if there is sufficient room in the scratchpad to store the values, if it is in sufficient the program would abort with a suitable error message indicating that the startup value for the scratchpad needs to be increased.

5.11 Implementation details of the Naïve Parallel In-Close3 Algorithm

Figure 5.19 shows how the recursive function `Parallel_ComputeConceptsFrom()` is called from the `main()` function. It is executed in the main master thread.

```
#pragma omp parallel
#pragma omp master
{
    Parallel_ComputeConceptsFrom(A, ASize, BBit, 0, 0, NBit, 0);
}
```

Figure 5.19 , OpenMP Initial parallelization code used to call recursive function from main program

Figure 5.20 shows the coding that was used to implement the lines 14 to 16 of the Naïve Parallel In-Close3 algorithm (See *Figure 5.1*). Line 8 of the Implementation (See *Figure 5.20*) shows the use of the omp task command that used to spawn a new process. The parameters that are used to send the recursive function in line 12 needs to be copied to duplicate variables (See line 3, 4) so that they can be persisted by OpenMP in an internal queue. The OpenMP

```
1  while (get(q, &C, CSize, tc, tj, level)) {
2     int *tC;
3     ALIGNED_VECTOR_TYPE(tBChildBit[VECTOR_MAX_COLS_CELLS]);
4     ALIGNED_VECTOR_TYPE(*tMBit[ATTRIBUTESIZE]);
5     tC = C;
6     memcpy(tBChildBit, BChildBit, nArray*VECTOR_SIZE_BYTES);
7     memcpy(tMBit, MBit, ATTRIBUTESIZE*sizeof(VECTOR_TYPE *));
8     #pragma omp task firstprivate(tC, CSize, tBChildBit,tj, tc,level,tMBit,tid) {
9         tid = omp_get_thread_num();
10        nodeParent[tid + 1][highC[tid + 1][0]] = c;
11        nodeParentID[tid + 1][highC[tid + 1][0]] = id;
12        Parallel_ComputeConceptsFrom(tC, CSize, tBChildBit, tj+1, highC[tid + 1][0]++,
13 tMBit, level);
   }
```

Figure 5.20, Recursive Call from Naive Parallel In-Close3 Implementation scheduling engine will execute line 12 only when certain conditions are met. This implies that several omp task requests can be made to OpenMP and this operation is asynchronous in nature. The `nodeParentID` array is a new array that was introduced to capture the current thread id (processor id). This enables the intents to be reassembled correctly from the BTree data structure.

```
 1 if (level == LEVEL) {
 2    int *tA;
 3    VECTOR_TYPE tBChildBit[VECTOR_MAX_COLS_CELLS];
 4    VECTOR_TYPE *tMBit[ATTRIBUTESIZE];
 5    tA = A;
 6    tid = id;
 7    memcpy(tBChildBit, BParentBit, nArray*VECTOR_SIZE_BYTES);
 8    memcpy(tMBit, NBit, ATTRIBUTESIZE*sizeof(VECTOR_TYPE *));
 9   // Capture the parameters of the function so that it can be called on later
10 #pragma omp task firstprivate(tA, ASize, tBChildBit,y, c,level,tMBit,tid) {
11       tid = omp_get_thread_num();
12       nodeParent[tid + 1][highC[tid + 1][0]] = c + 50000000;
13       ComputeConceptsFrom(tA, ASize, tBChildBit, y, highC[tid + 1][0]++,
14 tMBit, level);
15  }
16      return;
17 }
```

Figure 5.21 , Directly spawning parallel tasks in Direct Parallel In-Close3 Implementation

5.12 Implementation details of the Direct Parallel In-Close3 Algorithm

Figure 5.21 corresponds to the lines 2 and 3 of the Direct Parallel In-Close3 algorithm (See *Figure 5.8*). The OpenMP task command is used here as well. Since an incrementally generated intent can reside only on two processes there is no need to use a separate array to capture the process id. Instead the `nodeParent` array (See line 12) which is used to link the parent node to the child node has a unique number (50,000,000) added to the current concept number c. The algorithm that traverses the BTree easily identifies this and know that the initial part of the intent is in the master process.

5.13 Implementation details of the Queue Parallel In-Close3 Algorithm

```
if (level == LEVEL) {
#pragma omp critical {
    int thtemp = queItemNo % NO_PROCESSORS;
    put(que[thtemp], A, ASize, BParentBit, y, c + 50000000, NBit, level);
    queItemNo++;
  }
  return;
}
```

Figure 5.22 , Generating parallel tasks in Queue Parallel In-Close3 algorithms

The code in *Figure 5.22,* corresponds to line 2 and 3 of the Queue Parallel In-Close3 algorithm (See *Figure 5.9*). Here too the number (50,000,000) appears to handle the traversal of the BTree. It's used to indirectly connect to a child node which resides on a different process that of a parent node that resides in the main process. An explicit queue is used to capture the parameters to be sent.

The code in *Figure 5.23* corresponds to lines 19 to 21 of the Queue Parallel In-Close3 algorithm (See Figure 5.9). Here too the OpenMP task command is used for spawning the parallel process. The code in *Figure 5.23* spawns the tasks in a different order from that it was captured.

Finally, the code in *Figure 5.24* also corresponds to lines 19 to 21 of the Queue Parallel In-Close3 algorithm. Here lines 3 to 17 are run in parallel. This implementation does not use the OpenMP task command. Line 3 commences a pool of threads which are available and uses line 12 to access the specific queue assigned for a given thread. The parallel recursive process is spawned within the same thread.

```
if (level == 0) {
   // Spawn new threads for each queue, recursive call from queue
   for (int thID = 0; thID < NO_PROCESSORS; thID++) {
      // for each loop one each for each thread
      int *tA;
      int tASize;
      VECTOR_TYPE tBChildBit[VECTOR_MAX_COLS_CELLS];
      short int tlevel;
      int tc, ty;
      VECTOR_TYPE *tMBit[ATTRIBUTESIZE];
      while (get(que[thID], &tA, tASize, tBChildBit, ty, tc, tMBit, tlevel)) {
      #pragma omp task firstprivate(tA, tASize, tBChildBit,ty, tc,level,tMBit) {
          id = omp_get_thread_num();
          nodeParent[id + 1][highC[id + 1][0]] = tc;
          ComputeConceptsFrom(tA, tASize, tBChildBit, ty, highC[id + 1][0]++,tMBit,
tlevel);
      }
   }
}
```

Figure 5.23, Spawning parallel tasks in Simple Queue Parallel In-Close3 Implementation

```
if (level == 0) {
  // Spawn new threads for each queue, recursive call from queue
#pragma omp parallel {
   // for each loop one each for each thread
   int thID = omp_get_thread_num();
   int *tA;
   int tASize;
   VECTOR_TYPE tBChildBit[VECTOR_MAX_COLS_CELLS];
   short int tlevel;
   int tc, ty;
   VECTOR_TYPE *MBit[ATTRIBUTESIZE];
   while (get(que[thID], &tA, tASize, tBChildBit, ty, tc, MBit, tlevel)){
        nodeParent[thID + 1][highC[thID + 1][0]] = tc;
        ComputeConceptsFrom(tA, tASize, tBChildBit, ty, highC[thID + 1][0]++, MBit,
tlevel);
    }
}
}
```

Figure 5.24, Spawning parallel tasks in OpenMP Queue Parallel In-Close3 Implementation

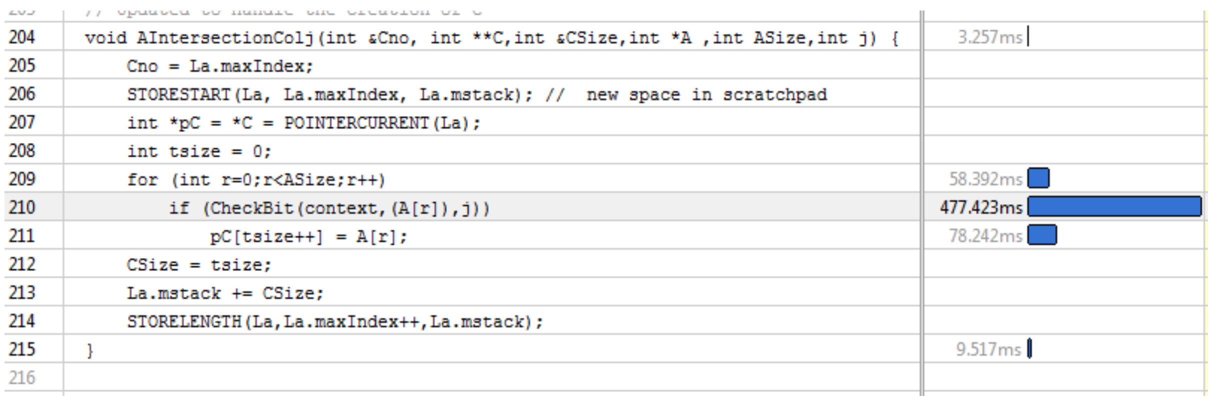
5.14 Optimising carried out in the parallel implementations

The serial In-Close3 implementation was optimised prior to parallelization efforts. The Intel® VTune™ Amplifier XE 2017 was used to identify hotspots which are the parts of the code that consumes the most time during execution. These are the parts the code where optimisation will bring the biggest impact on performance.

Table 5.9, VTune™ Amplifier XE 2017, Hotspot analysis of the serial version of In-Close3

Function	**Module**	**CPU Time**
IsNjSubSetofBuptoJBit	CBO5.exe	0.667s
AIntersectionColj	CBO5.exe	0.627s
ComputeConceptsFrom	CBO5.exe	0.392s
BuptoJisEqualtoPartialClosureOfCuptoJBit	CBO5.exe	0.325s
malloc	ucrtbased.dll	0.188s
[Others]	N/A*	0.367s

The hotspot analysis given in *Table 5.9,* shows that the functions `IsNjSubSetofBuptoJBit()` and `AIntersectionColj()` take the most time in the implementation.



```
void AIntersectionColj(int &Cno, int **C,int &CSize,int *A ,int ASize,int j) {      3.257ms
    Cno = La.maxIndex;
    STORESTART(La, La.maxIndex, La.mstack); //  new space in scratchpad
    int *pC = *C = POINTERCURRENT(La);
    int tsize = 0;
    for (int r=0;r<ASize;r++)                                                       58.392ms
        if (CheckBit(context,(A[r]),j))                                            477.423ms
            pC[tsize++] = A[r];                                                     78.242ms
    CSize = tsize;
    La.mstack += CSize;
    STORELENGTH(La,La.maxIndex++,La.mstack);
}                                                                                    9.517ms
```

Figure 5.25 , Hotspot Analysis of the Serial InClose3, AIntersectionColj() function

Figure 5.25, shows the Hotspot Analysis of the `AIntersectionColj()` function. We can also clearly see that the `CheckBit()` function occupies the most amount of time. Vladimmirov describes in detail strategies that can be under taken to optimize serial code to parallel (Vladimirov & Karpusenko, 2013). These are general instructions that will work well for all types of Intel processors including the Intel Xeon Phi many core processors. The key strategies that worked in the optimization of the serial code are data aligning, and using vectorization. Data aligning is the process of padding structures so they match the default word sizes of the processors. Vectorization is the process in which a compiler can generate instructions to work on multiple data items at once in parallel for certain arithmetic operations. For instance, modern Intel Xeon Processors support AVX2 instructions which support 256 bits. This implies if there is a loop that uses a matrix manipulation that involves 32 bit integers then 256/32 number of integers can be manipulated by one vector instruction. Due to variation in vector based assembly instructions a programmer would specify the intent of vectorization using compiler directives such as `#pragma simd` or force vectorization using directives such as `#pragma ivdep` (Chappell & Stokes, 2012; Vladimirov & Karpusenko, 2013).

Once parallelization was carried out by using OpenMP, the VTune™ Amplifier XE 2017 was used to see the occupancy of the cores during parallel execution. OpenMP has a scheduling algorithm which handles the parallelization. This can be tuned using directives. The static, 1 setting produced the best parallelization results.

Table 5.10, VTune™ Amplifier XE 2017,analysis of Paralel Impleentation

CPU Utilization:	16.0%
Average CPU Usage:	0.641 Out of 4 logical CPUs
Memory Bound:	21.3%
Cache Bound:	0.219
DRAM Bound:	0.005

Table 5.10 shows an analysis of running a parallel version of In-Close3 on a Windows Laptop. FCA algorithms by nature are memory bound as the computation involved can be handled by bitwise operations. In memory bound applications the bottleneck is the memory access, in essence the CPU is starving for data to reach its registers from memory.

5.15 Discussion and Conclusion

The current codebase has further room for optimization by removal of mutexes which are locks placed when accessing shared memory variables. The use of mutexes can be limited. False memory sharing is a phenomenon that occurs when multiple cores accesses data which are close by, for instance array elements which are close by. Due to caching these adjacent values are loaded to multiple cores, which causes caches to be refreshed during parallel execution when one core updates one element. This can be easily avoided by padding data or ensuring that there is a gap between the data items being processed simultaneously (Vladimirov et al., 2015).

What are the options for parallelizing the chosen serial algorithm (e.g. shared memory, distributed solutions) and which options may be the best in terms of speed and scalability.

This chapter introduced three new shared memory parallel algorithms Direct Parallel In-Close3, Queue Parallel In-Close3 and Naïve Parallel In-Close3. The limitations of the Naïve Parallel In-Close3 algorithm was shown using the scalability graphs presented in *Figure 5.3*. The Direct Parallel In-Close3 and Queue Parallel In-Close3 algorithms scale well and the scalability graphs are shown in *Figure 5.10* and *Figure 5.11*. The Queue Parallel In-Close3 algorithm was implemented using strategies OpenMP Queue Parallel In-Close3 and Simple Queue Parallel In-Close3. In general the Direct Parallel In-Close3 implementations gave the overall best results.

6 Distributed Memory Parallel Algorithms

6.1 Distributed Memory Machines

Distributed Memory Architecture is a common type of Parallel Computing Architecture. The simplest form of creating a distributed memory architecture type parallel machine is to connect multiple computers together through a network.

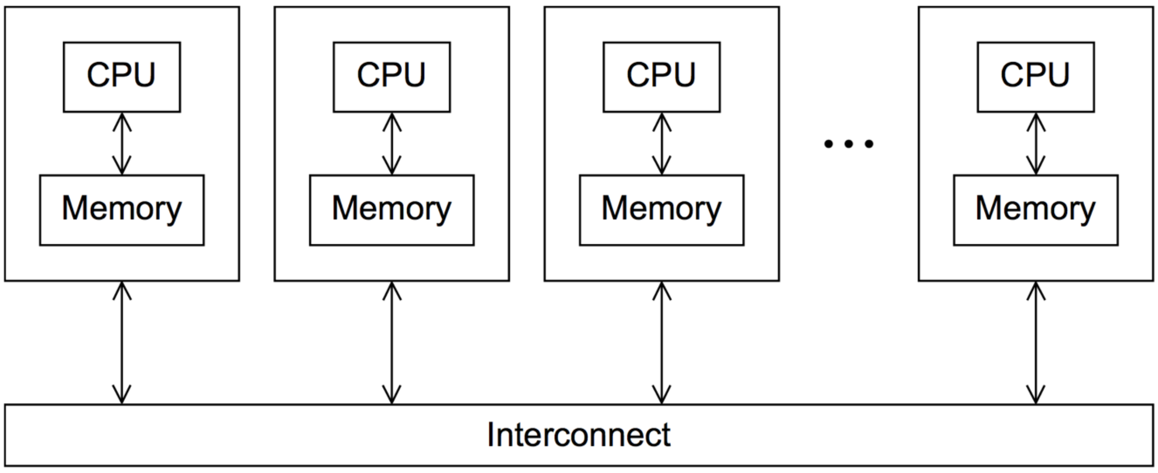


Figure 6.1, Distributed Memory System (P. Pacheco, 2011)

Figure 6.1, Shows a typical distributed memory architecture type parallel machine configuration (Pacheco, 2011). Each CPU has a separate Memory, which is not shared. The term Cluster is used to describe such an arrangement of computers. According to Fynn's classification clusters are Multiple Instruction Multiple Data (MIMD) machines (Flynn, 1966). Each CPU can run different instructions working on different data. In Cluster computing each computer connected through a high speed network is a commodity computer. The computers that forms the cluster runs a cluster based operating system, which allows the computers to work in unison to run parallel applications.

Purpose built cluster computers are assembled by combining stripped down versions of servers which are called blade servers. Each blade server is built using a modular design and optimizes the space and power consumption. A blade server is fitted to a blade enclosure that can accommodate multiple blade servers. A typical blade server may have multiple CPUs attached. A CPU is typically named as a Node (Mani & Jee, 2007).

Figure 6.2, shows how a parallel application runs on top of a cluster middleware. A cluster based operating system provides the user of the cluster a unified view. Typically each server runs a stripped down version of an operating system that is optimized for performance (Silva & Buyya, 1999). A majority of vendors including super computers now prefer a customized version of linux distribution for this purpose (Strohmaier, Dongarra, Simon, & Meur, 2018).

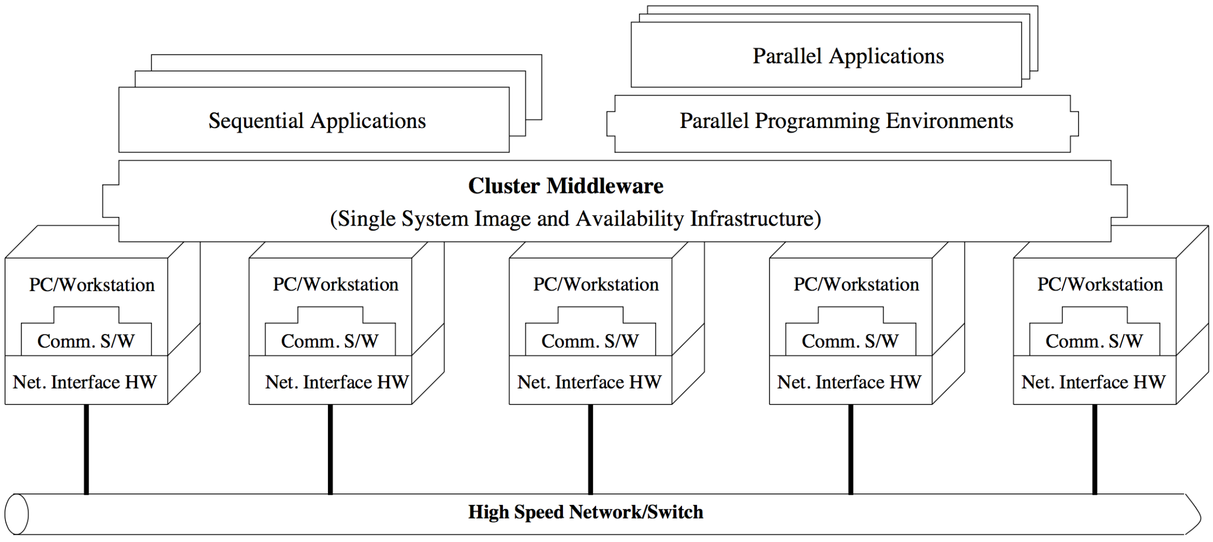


Figure 6.2,Cluster computer architecture (Silva & Buyya, 1999)

Figure 6.3, shows the different components and users who interact with a cluster. A user typically remotely logs into the cluster and submits an application to run to a Batch Scheduler. The scheduler is responsible for prioritizing the jobs in the queue and to allocate resources in the cluster to the application. A typical job submission will contain the parallel application executed, location of the data files and the number of nodes that should be utilized in running the application. In a heterogeneous cluster it will be possible to specify the specific nodes that are needed to execute the parallel application (Hussain et al., 2013).

High Performance Computing (HPC) is an area in computer science where scientists and engineers use supercomputers to solve complex computationally heavy applications in a multitude of fields. Typical application scenarios include simulation of natural phenomena using numerical methods. Super Computers are specialized clusters that are highly optimized for computations and can perform computations exceeding petaflops (10^{16} floating point operations per second) (Saecker & Markl, 2013).

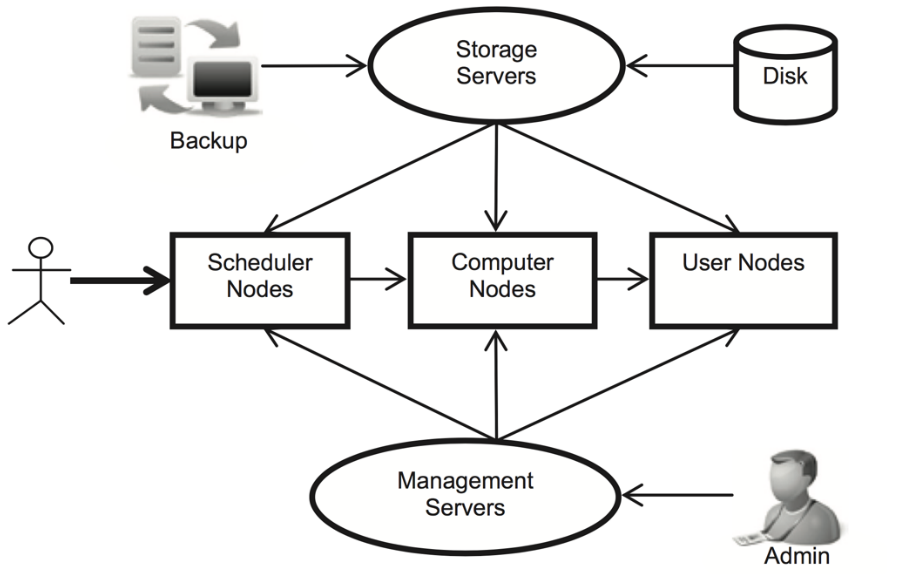


Figure 6.3,A cluster computing system architecture (Hussain et al., 2013)

6.2 Distribued Memory Parallel Programs

One general approach that we can use to program such a system is to write separate programs that run on different CPU's. Each program carries out a specific task similar to members of a project team that work independently on different aspects of the project. If the team is small the project leader can individually assign the work to each of the members of the project. However, this arrangement is not possible to manage if the team grows to a larger number. A simple scalable approach the project manager can adopt is to keep a generated list of tasks that needs to be completed and randomly assign it to members.

Due to scalability and performance reasons, distributed memory computers typically adopt a similar strategy of the project manager. Single Program Multiple Data (SPMD) is the technique used to achieve parallelism where all nodes running the parallel application run the same program, where the main node (master) distributes work to the other nodes (workers) based on their availability.

These programs would need to communicate the work to be assigned, and the results computed by sending messages similar to how the project members who would use email as their communication mechanism.

6.3 Existing Parallel FCA Algorithms

Huaiguo Fu had created a parallel implementation of the NextClosure algorithm (Fu & Foghlu, 2008; Fu & Nguifo, 2004). Krajca (Krajca, Outrata, & Vychodil, 2008) presented a parallel algorithm called PFCbO which parallelizes the FCbO algorithm. There have been several attempts to develop distributed parallel FCA algorithms. Krajca and Hu have presented Map Reduce versions of FCA algorithms (Krajca & Vychodil, 2009; Xu, de Fréin, Robson, & Foghlú, 2012). Kengue has developed a MPI implementation for Closed Itemsets and Implication Rule Bases (Kengue, Valtchev, & Djamegni, 2007).

6.4 Proposed Distributed Memory Algorithms

The key criteria that was considered in the technology to be used in developing the distributed parallel algorithm was the selection of an implementation that produces the fastest computational speeds. Today's Big Data buzz word technologies such as Apache Spark which is 100 times faster than Hadoop, is still extremely slow compared to MPI which is the defacto programming standard in High Performance Computing (HPC) (Reyes-Ortiz, Oneto, & Anguita, 2015).

Mattson, details Patterns that can be used for distributed parallel computing. These patterns are best practices that have been developed in the industry. The Master Slave Pattern using the SPMD model is ideal for the implementation of recursive algorithms (Mattson, Sanders, & Massingill, 2004). Figure 6.4 shows the flowchart of the Master/Worker Pattern. One of the processors in the distributed system acts as the Master and all other processes act as workers. The master process computes a set of tasks that need to be computed. These are delegated to each worker that is available until all tasks are carried out. In the implementation, all the processors run the same program. Each processor is identified by a unique number called the rank. The parallel program has different sections for the Master Process and the Worker Process based on the rank.

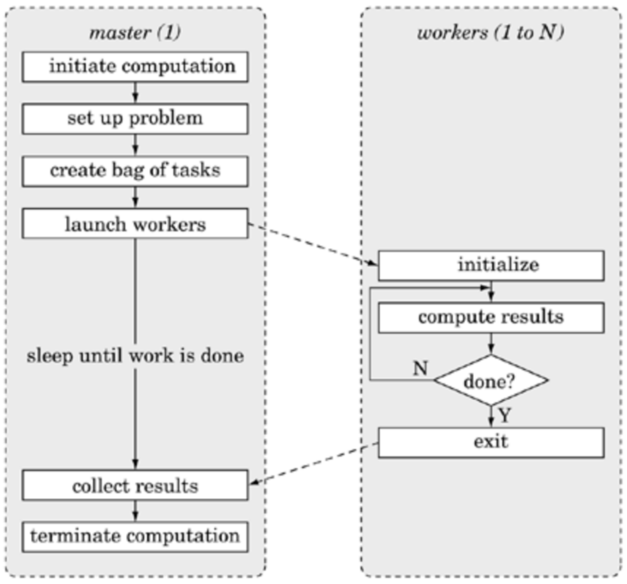


Figure 6.4, Master Worker Pattern (Mattson, Sanders, & Massingill, 2004)

The flowchart shown in *Figure 6.5* gives a high-level view of how the distributed version of In-Close3 algorithm is implemented. The solution involves the use of one Master Node, one Result Node and multiple Worker nodes. When parallelizing a serial algorithm, a shared memory implementation requires less refactoring compared to a distributed memory implementation. This is partly due to how one writes shared memory programs using a library such as OpenMP compared to a distributed parallel programming library such as MPI. The major difference in distributed memory parallel programs is that data and results cannot be shared among the different processors directly using the same variables.

In parallelizing In-Close3 initially the shared memory algorithm and implementations were developed. Subsequently a distributed memory parallel algorithm and implementation was developed. In transforming the shared memory algorithm to the distributed memory algorithm it was decided to spawn worker nodes as soon as parallel task was identified. Due to this decision, the Master Process will serially compute a set of parallel tasks and may continue to still work on the generation, when a Worker completes the allocated task. A Result Process is introduced to capture incoming results from workers after their allocated task is completed. In the algorithm presented only one Result Process was used, this could however be extended to multiple nodes as part of an optimizing process.

The master node reads the context file which contains the data to be processed and broadcasts this to all nodes in the distributed parallel computer. This is then broadcasted to all the workers as a message which allows each of the worker nodes to keep a local copy of the entire context which is needed in the computation. *Figure 6.16*, shows a visual representation of how data is transmitted between the node. The think lines represent the one off broadcast sent from the Master Node to the Worker nodes to transfer the dataset (context). The dashed lines from the Master Node to the Worker nodes represents the parameters sent to invoke the recursive sub tree computation of concepts. Finally the Worker to Result Node dashed lines show the flow of the concepts generated from the workers to the result node. This is further described in Section 6.5. After the broadcast has been done the master node starts computing concepts starting from the root of the recursive call tree. The decision to send part of the recursion sub tree to a worker is based on a parameter (LEVEL) which keeps track of the recursion level. A similar approach was used in the shared memory algorithm as well. When a suitable task which can be parallelized (i.e. a recursive call subtree) is found, the task is sent as a message to a Worker Node who is ready to undertake the given task. The Master Node continues to do this until it has exhausted all parallel tasks that need to be sent.

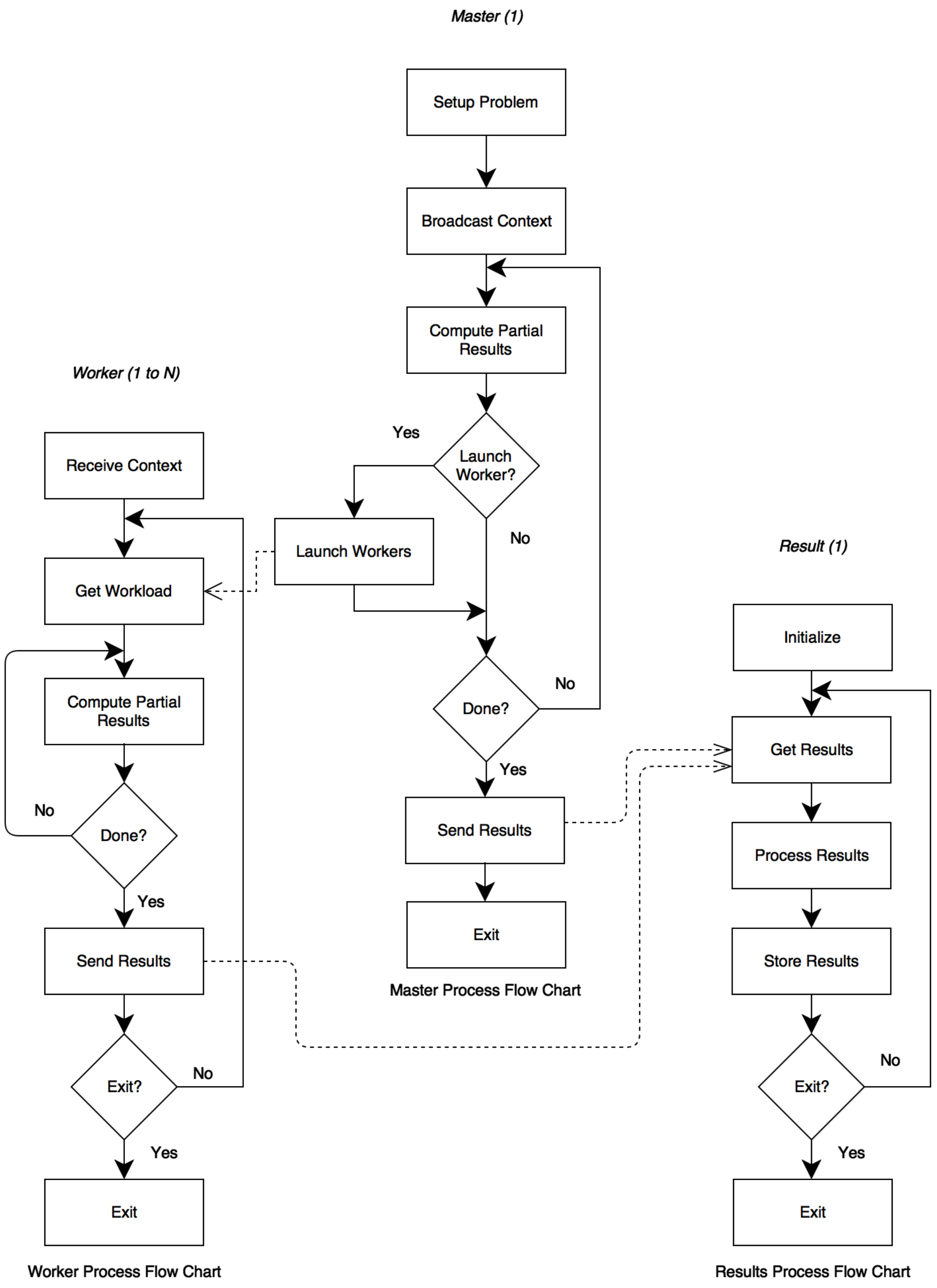


Figure 6.5, Flowchart of Distributed Memory Parallel In-Close3

A worker who gets a message containing a task that needs to be computed, starts computing the concepts until all the concepts which come under the specific recursive call sub tree is completed. The result set are sent to the Results Process. The Result Process takes the result set obtained and stores the results to disk.

Figure 6.6 shows detail pseudo code of the distributed memory parallel In-Close3 algorithm. The main function listed in *Figure 6.6* shows how the Master Process reads the context and broadcasts it to all the nodes available. It also shows how the Master Process, Worker Processes and the Results Process executes their respective functions `MasterProcess()`, `WorkerProcess()` and the `ResultProcess()`.

The `MasterProcess()` function described in *Figure 6.6* (see line 17), initially calls the In-Close3's `MainRecursiveFunction()` (see line 54) with details of the first concept and recursion level zero. Once the main recursive function has completed the computation the final results are sent to the Result Process using the `SendResultMessage()` function. Next the `MasterProcess()` iterates through all workers to check if they are ready to accept a task and then sends a message to terminate the Worker. Finally, a message is sent to the Result Process to terminate the Result Process.

```
int function main()
   // mpi initializations
   if master then
      read context
   endif
   broadcast context
   if master then
      MasterProcess()
   elseif worker then
      WorkerProcess()
   elseif resultsprocess then
      ResultsProcess()
   endif
   // mpi closures
end

void function MasterProcess()
    MainRecursiveFunction(parameters,0)
    SendResultMessage()
    // after we have got all child results (they may be still working)
    foreach worker
       readmessage
       if ready then
          sendmessagetoworker to quit
       endif
    endforeach
    sendmessagetoresultsprocess to quit
end

void function WorkerProcess()
   while (true)
      SendMessage(ready)
      ReceiveMessage();
      if workload then
         WorkerRecursionFunction()
         SendResultMessage() // concatnated results
      elseif exit then
         break;
      endif
   endwhile
end

void function ResultsProcess()
    while (true)
       SendMessageToProcess(ready)
       ReceiveMessageResults();
       if results then
          ProcessResults()
       elseif exit then
          break;
       endif
    endwhile
end
```

Figure 6.6(a), Pseudocode - Part 1 - Distributed Memory Parallel In-Close3

```
54 void function MainRecursiveFunction(parameters,
55         recursionlevel)
56    if recursionlevel = LEVEL then
57       SendMessagetoWorker(parameters)
58       return
59    endif
60    // body of recursive function
61    if isCanonical()  then
62       MainRecursiveFunction(parameters, recursionlevel+1)
63    endif
64    storeResults()
65 end
66
67 void function WorkerRecursionFunction(parameters)
68    if isCanonical() then
69       WorkerRecursionFunction(parameters)
70    endif
71    storeResults()
72 end
73
74 bool function isCanonical()
75    if concept is new then
76       return true
77    else
78       return false
79    endif
80 end
81
82 void function SendMessagetoWorker()
83    ReadMessagefromWorker()
84    if workerready then
85       SendMessage()
86    endif
87 end
```

Figure 6.6(b), Pseudocode - Part II - Distributed Memory Parallel In-Close3

The `MainRecursiveFunction()` described in *Figure 6.6* (See line 54) accepts the parameters of the concept being handled and the recursion level as parameters. When the recursion level reaches the predetermined LEVEL the details of the parameters which is effectively the starting point of the recursion sub tree will be sent to a Worker Process using the `SendMessageToWorker()` function. This function which is also in *Figure 6.6*, initially sends a message requesting for a worker process who is ready to respond and sends a message containing the recursive subtree to the worker process.

Each worker runs the `WorkerProcess()` function which is shown in line 30. This process runs in an infinite loop. Inside the loop a worker initially sends a message to the master process indicating that it is ready to accept a task. Once it receives a message from the master process, it checks if the message contains a task to be processed or whether it is a termination signal. If the worker has received a task to process it invokes the `WorkerRecursiveFunction()` in line 35. The `WorkerRecursiveFunction()` described in line 67 is similar to the `MainRecursiveFunction()` with the exception that it processes all the concepts that comes under the purview of the recursion subtree. The `WorkerRecursiveFunction()` is recursively called in line 69 based on the cannocity test. The `isCanonical()` function in line 74 essentially is used to check if a new concept has been generated. If the concept is new this function returns true otherwise it returns false. At the end of computing all the concepts that it can compute, the `WorkerRecursiveFunction()` calls `StoreResults()` in line 71 to send the results to the Result Process.

In the `MainRecursiveFunction()` (See line 54) for concepts at the higher level of the recursion subtree concepts are computed similar to the `WorkerRecursiveFunction()`. The recursion occurs in line 62 based on the same cannocity test.

The Result Process runs the `ResultProcessorFunction()` (see line 67) which is shown in line 43. Like the Worker Process this too runs in an infinite loop. First it sends a message to the workers and the master process that it is ready to receive a result set. Once it receives a list of results from a worker it processes the results, saves them to the hard disk and gets ready to receive another result set. The Result Process can also receive a result set from the master as well. Finally the master process will send a terminate signal which the Result Process will use to wrap up operations.

```
// Parameters for recursive function
typedef struct {
        int A[OBJECTSIZE];
        int ASize;
        VECTOR_TYPE BParentBit[VECTOR_MAX_COLS_CELLS];
        int y;
        int c;
        short int level;
        int flag; //  (TRUE -  valid, FALSE (exit))
        VECTOR_TYPE NBitFlag[VECTOR_MAX_COLS_CELLS];
        VECTOR_TYPE NBit[(VECTOR_MAX_COLS_CELLS) * ATTRIBUTESIZE];
} tParaCompound;
```

Figure 6.7, User defined type representing the Recursive Algorithm Parameters

```
void WorkerComputeConceptsFrom(int A[], int ASize, VECTOR_TYPE *BParentBit,
          int y, int c, VECTOR_TYPE *NBit[],short int level )
```

Figure 6.8, WorkerComputeConceptsFrom() function prototype

In distributed memory parallel algorithms, the parameters of functions that are spawned in parallel needs to be serialized into messages and sent. The parameters in the `WorkerComputeConceptsFrom()` function shown in *Figure 6.8* needed to be sent from the Master Process to the worker processes. A C structure which contains an aggregation of all the parameters called `tParaCompund` was defined (See *Figure 6.7*). The serialization process involves copying all the parameters into this C Structure. The `MPIPackData()` function listed in the Appendix *Figure B.1* shows in detail how the parameters in the Master Process are copied to a variable of the `tParaCompund` structure type. Section B.1 describes in depth how the data structures needed for message passing are defined in MPI and how the data is packed

```
// Master Process Sends Message with Tasks to spawn to the Worker Processes
void SendMessageToWorker(tParaCompound &msg, const int *Av, const int &ASize, const VECTOR_TYPE
*BParentBit, const int &y, const int &c,  VECTOR_TYPE *NBit[], const short int &level) {
   MPI_Status status;
   if (ReceiveWorkerMessage(status)) {
        MPIPackData(msg, Av, ASize, BParentBit, y, c, NBit, level);
           mpidebugflag = false;

        MPI_Ssend(&msg, 1, TaskType, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
   }
}
```

Figure 6.9, SendMessageToWorker() function

(serialized) and unpacked (de-serialized) by the sender node and receiver node respectively. The `SendMessageToWorker()` function listed in *Figure 6.9*, shows how the parameters that needs to be sent to the worker processes are copied to a variable of the `tParaCompound` C data structure and sent to workers using the `MPI_Ssend()` MPI method. Initially the `SendMessageToWorker()` function gets details of a worker process that is ready to accept a task through the `ReceiveWorkerMessage()` function which is shown in *Figure 6.11*.

```
1 // Master getting a message from a worker that he is ready
2 bool ReceiveWorkerMessage(MPI_Status &status) {
3    int flag;
4    MPI_Recv(&flag, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
5    if (flag)
6       return true;
7    else
8       return false;
9 }
```

Figure 6.10, ReceivePayload() function

The `MPI_ANY_SOURCE` parameter used in line 4 of the `MPI_Recv()` MPI function in *Figure 6.11*, allows the Master Process to select any one of the free Worker Processes. The `status` parameter which it returns captures the rank of the selected worker process. A message that the Master Process receives with the flag set to true, is a validation to check that the sender is truly readly to accept a workload. Only a worker process will send a message to the master process indicating that it is ready to accept a new workload. In the `SendMessageToWorker()` function the `MPIPackData()` function does the actual copying of the parameters to the C Structure. The `NBit[]` array which is used to capture the canonicity test failures can contain blank values.

The `MPIPackData()` function, shown in the Appendix B copies the parameters used in the Master Process to the C++ structure parameter called `msg` which is of the `tParaCompound` type. The `ReceivePayload()` function shown in *Figure 6.10*, calls the `MPIUnPackData()` function to de-serialize the data that it receives.

```
// Worker Processes Receive Payload from the Master Process with task details to
// spawn or a Kill Message from the Master Process
bool ReceivePayload(int *Av, int &ASize, VECTOR_TYPE *BParentBit, int &y, int &c, VECTOR_TYPE
*NBit[], short int &level) {
        tParaCompound msg;
        MPI_Status status;
        MPI_Recv(&msg, 1, TaskType, MPI_ANY_SOURCE, MASTER, MPI_COMM_WORLD, &status);

        bool flag = MPIUnPackData(msg, Av, ASize, BParentBit, y, c, NBit, level);
        return flag;
}
```

Figure 6.11, ReceivePayload() function

All valid messages that contain data are sent with the flag set to true. The two kill process functions `SendKillMessageToWorker()` and `SendKillMessageToResults()` sends a message with the flag set to false (See *Figure 6.12*). This is checked by the main loops running the worker and results processes (See *Figure 6.6*). These functions are called by the Main Process once all the computations have been completed.

In the current implementation the Result Process dumps the computed results (concepts) to the disk enabling the Master Process to finally assemble and display all the computed concepts.

The Distributed Parallel In-Close3 implementation was tested on the Archer Super Computer[1]. Each Node has two Intel Xeon E5-2697v2 processors connected with each processor having 12 cores. A single node has 64GB of RAM. The graphs containing the performance results of running the real world datasets Mushroom, Adult and Internet Ad are shown in *Figure 6.13,*

[1] http://www.archer.ac.uk

Figure 6.14. The peak performance obtained varied with the datasets used. Mushroom gave the best results at 18 cores, Adult at 12 cores and Internet Ad at 6 cores. A detail expanded version of the experiments carried out for the Mushroom dataset is presented in *Figure 6.15*.

We can see clearly that the performance of the Mushroom dataset increases upto 18 cores and then it subsequently decreases.

```
void SendKillMessageToWorker(const MPI_Status &status) {
   tParaCompound msg;
   msg.flag = false;
   MPI_Ssend(&msg, 1,TaskType, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
}

// Master Process sends the kill Message to the Results Process
void SendKillMessageToResults() {
   int position = 0;
   int flag = false;
   MPI_Pack(&flag, 1, MPI_INT, resultsBuffer,4, &position, MPI_COMM_WORLD);
   MPI_Ssend(resultsBuffer, position, MPI_PACKED, RESULTS, 0, MPI_COMM_WORLD);
}
```

Figure 6.12 , SendKillMessageToWorker() and SendKillMessageToResults() functions

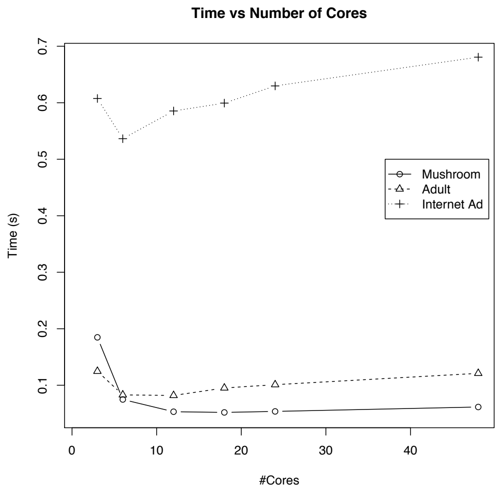


Figure 6.13, Real World data sets, Distributed Memory Parallel In-Close3 Time vs Processors

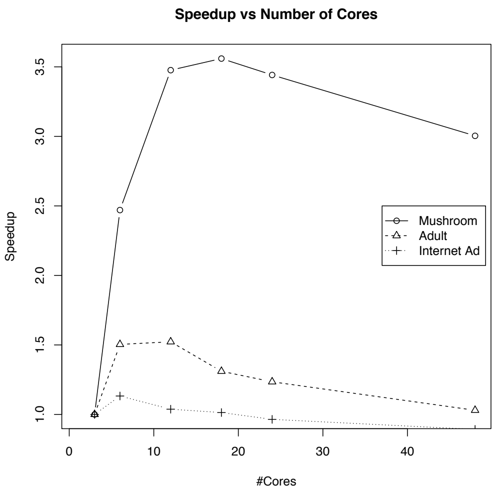


Figure 6.14, Mushroom, Distributed Memory Parallel In-Close3 Time vs Processors and Speedup vs Processors

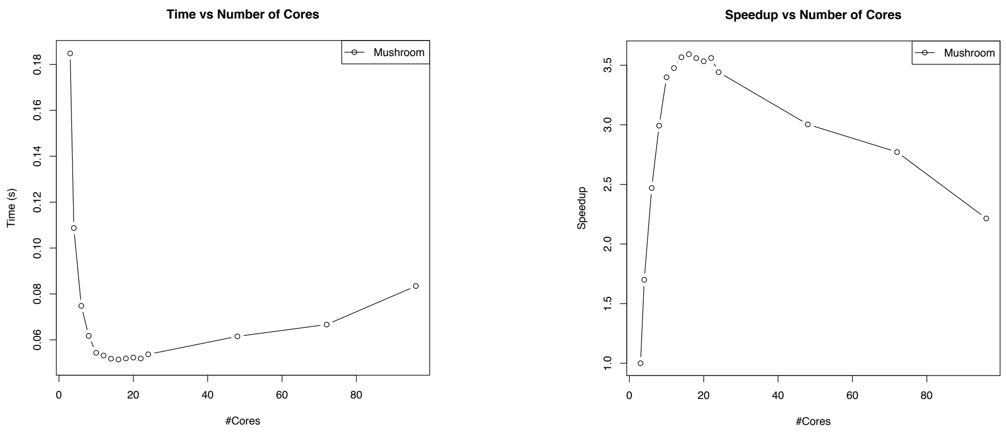


Figure 6.15, Mushroom, Distributed Memory Parallel In-Close3 Time vs Processors and Speedup vs Processors

Table 6.1 shows the running times of the mushroom dataset for different processor configurations. From the above results, we can see that the implementation runs without issues up to 48 physical processors simultaneously. In the above example each processor was loaded upto two cores only. There is no significant variation in the timing given the same number of cores. The overhead due to message passing is seen with the increase of the number of cores.

Table 6.1, Running Times of Mushroom of Distributed Parallel In-Close3 on the Mushroom dataset

Nodes	Processors	Cores	Time (s)
01	02	24	0.0537
12	24	24	0.0540
02	04	48	0.0615
12	24	48	0.0589
24	48	48	0.0593
03	06	72	0.0667
04	08	96	0.0834
24	48	96	0.0737

Table 6.2 shows a comparison of the distribution of running the Direct Parallel In-Close3 implementation and the Distributed Parallel In-Close3 Implementation running on one compute Node in the Archer super computer. In both instances all the cores were utilized. We can see that the Distributed Memory Implementation performs better than the shared memory Implementation in this specific instance where the Mushroom dataset is computed.

Table 6.2, Comparison of the Shared Memory implementation (Direct Parallel In-Close3) and the Distributed Parallel In-Close3 Implementations running on the Archer Super Computer Intel Xeon E5-2697v2

Implementation	Nodes	Processors	Cores	Time (s)
Shared Memory	01	02	24	0.1135
Distributed	01	02	24	0.0537

6.5 Discussion of the Distributed Memory implementation of In-Close3

6.5.1 Experiments carried out

The main outcome of developing the distributed memory algorithm was to develop a functioning implementation. *Table 6.1* clearly shows that the implementation scales well upto even 24 nodes, which means that the distributed implementation ran on 48 processors simultaneously. The mushroom dataset is a real word dataset.

The implementations were tested on the UK National Super computer Archer[2] which is a Cray XC30 which has a Cray Aries interconnect connecting 4 Nodes. The MPI latency using the Cray Aries interconnect is 1.3 micro seconds. When using more that 4 Nodes the optical cables with a latency of 100 nano seconds needs to be used. The hard disk space allocated to the compute nodes is a high performance parallel luster file system.

The experimental setup for getting timing results is described briefly. *Figure 6.16*, shows a visual representation of the execution of the distributed memory implementation of parallel In Close3. The double arrows show the secondary storage Input/Output operations and the single arrow from Master to Worker shows the single broadcast operation where the entire dataset (context) is copied to all worker nodes. The dashed arrows represent the data message that are sent from the Master Node to the Worker nodes and the Worker nodes to the Result nodes. The messages sent from the Master Node to the Worker Node contain the parameters sent to the Worker to execute part of the recursive sub tree. The Worker to Result node contain the generated concepts.

[2] http://www.archer.ac.uk

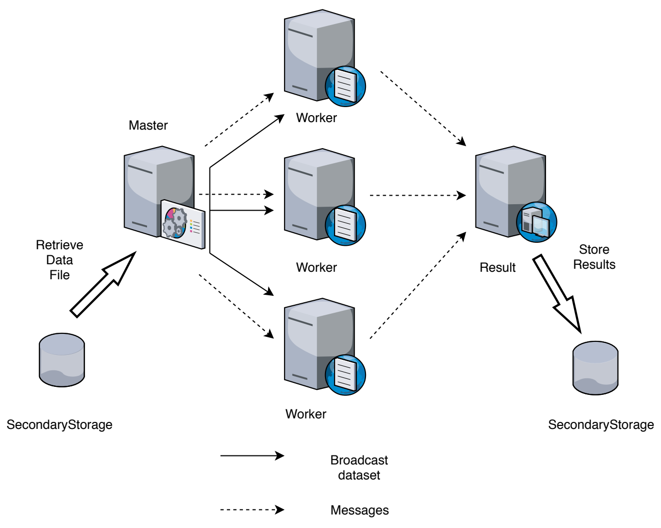


Figure 6.16, Visual Representation of the execution of the distributed memory implementation of parallel In-Close3

To be consistent with the serial and shared memory timing results the Secondary Storage input/output operations were excluded. Before the algorithm can start execution properly, all worker nodes need to have a complete copy of the context. The context is broadcast by the Master Node to all the worker nodes at the beginning. This too was not considered as part of the timing as both the serial and shared memory experiments, the time taken to load the context to memory was not considered. For timing purposes, the results were not stored to in the secondary storage.

One of the major reasons for non-deterministic behaviour for a distributed system are input/output operations and network operations. By eliminating the disk I/O operations and broadcast operations, the experimental results show less variability.

Each worker node gets a complete copy of the context at the very beginning and requires only the specific parameters needed to compute a specific sub tree of the recursive solution. Technically the algorithm running inside the worker node could be any FCA algorithm which can compute the concepts for the given sub tree. It requires no further interaction with other nodes until all the concepts in the recursive sub tree is computed. Thus, the worker node should contain the fastest serial implementation of generating concepts. Serial implementations of the Next Closure and other simpler algorithms were found to be slower than the serial CbO based algorithms.

6.5.2 Validation and Testing of the Implementation

The key aspect of developing distributed memory solutions is to initially ensure that the implementations work 100% correctly serially. After message passing is introduced to send data across nodes, unit tests need to be carried out to ensure that the data is marshalled correctly from the sender to the receiver. Unlike web service calls, where one doesn't need to worry about packing and unpacking data, MPI requires the developer to pack and unpack the data that is sent across nodes. Minor mismatches on the data formats of the messages is a common cause of errors.

Debugging parallel programs is extremely difficult compared to serial programs. Distributed memory applications has its own unique set of challenges to identify the cause of errors. When a program crashes a set of log print statements may help identify the statement that caused the system to break. The Cray ATP (Abnormal Termination Processing) was used to identify the exact location that caused a program to crash. Once enabled the tool generates a report which can be used to infer the root cause for the program crash.

Once the implementations were producing results without any errors, a similar approach used in serial programs was used validate the results. The output of the complete concept listing was compared with known results using the linux diff utility.

6.5.3 Optimization of the Implementation

The distributed Memory Parallel implementation of In-Close3 can be optimized by logging the performance of each of the specific workers. In the current implementation only one node is assigned to act as the Result Worker, this could be a bottleneck with the increase of computational nodes. The code can be easily refactored to take into account multiple Result Worker. Another major bottleneck in distributed memory algorithm implementations is the message passing. The master process, worker process and result process should ideally be able send messages with minimum delays. An analysis of a log containing the timing of messages which are sent through and forth different processes could reveal insights of potential bottlenecks.

A hybrid solution where a shared memory version of the algorithm runs in the worker node can also be considered to improve performance. However hybrid solution consisting of distributed and shared memory algorithms is in general complicated to optimize (Kedia, 2009).

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

7.1.1 Major Contributions of the Thesis

The major contribution of the thesis was the presentation of seven new FCA based algorithms. The three parallel algorithms Distributed Parallel In-Close3, Queue Parallel In-Close3 and Direct Parallel In-Close3 are the significant algorithms that are described. The Naïve Parallel Algorithm demonstrated the simplest approach in parallelizing a recursive algorithm. However its scalability is limited due to the less amount of computation each processor has to perform. The three serial algorithms presented in Chapter 4 were useful to explore all permutations of the three major enhancements to the CbO family of algorithms.

The implementation of the three parallel algorithms showed scaling with increase number of processors. However the distributed parallel In-Close3 algorithm should show the highest level of scalability. It was found that the algorithms by nature were memory bound, this is due to the fact that the computations required to generate concepts were not significant compared to compute bound problems. In general memory bound algorithms do not scale well compared to compute bound problems.

The implemented shared memory algorithms were compared with PFCbO and found to be faster. The serial and shared memory implementations were run on multiple computer configurations including a Laptop, High-end Cluster and a Super Computer. Similar results were obtained all three configurations validating the experiments carried out didn't depend on the Hardware.

7.1.2 Analysis of Serial CbO based algorithms

This thesis presented a thorough investigation of the state-of-the-art serial Formal Concept Analysis (FCA) algorithms. The CbO family of FCA algorithms were selected and the core features of the variations of existing CbO algorithms were isolated and combined in different permutations to analyse each of the core features and the algorithms. Both empirical and theoretical analysis were carried out for the eight different CbO variants that were considered. This included three new algorithms CbO-FC-DBF, CbO-FC-ICF-DF and CbO-PC-ICF-DF. These combined the three core features, combined depth and breadth search, partial closures and the inheritance of failed canonicity tests in a unique way compared to existing CbO algorithms. For the empirical analysis all the eight CbO variants were developed in a level playing field ensuring changes in the implementations were only due to the variations in the high level algorithm. Out of the three features considered there is a significant performance improvement when partial closure with incremental closure of intents is used in isolation. However, there is no significant performance improvement when the combined depth and breadth first search or the inherited canonicity test failure feature is used in isolation. The inherited canonicity test failure combined with the combined depth and breadth first feature produces better performances. It was also observed that partial closure with incremental closure of intents combined with depth and breadth first search also produces positive results. Partial closure with incremental closure of intents combined with the inherited canonicity test failure feature also produces positive results. Both empirical and theoretical analysis that was carried out confirmed that the best performance was gained by combining the three core features. This is essentially the CbO-PC-ICF-DBF algorithm presented in Chapter 4 which is Andrews In-Close3 algorithm.

7.1.3 Parallelization of In-Close3

The In-Close3 algorithm was the basis for the parallelization effort. Both shared memory and distributed memory parallel solutions were explored. This resulted in three shared memory parallel In-Close3 algorithms and one distributed memory parallel In-Close3 algorithm. All the proposed algorithms were implemented in a level playing field and tested empirically making use of the UK National Super Computing Service Archer[1] and Colfax Clusters[2].

The results demonstrate that CbO based algorithms which are naturally recursive by nature, can be easily parallelized with only minor changes to the codebase. OpenMP tasks can be used for this purpose where an entire recursive call sub tree can be assigned to separate threads.

Out of the three shared memory parallel In-Close3 presented the direct parallel In-Close3 and OpenMP queue parallel In-Close3 provided the best performances. The scalability of the parallel algorithms depended on the dataset being used.

[1] http://www.archer.ac.uk

[2] https://colfaxresearch.com

7.1.4 Analytical Analysis of Serial CbO based Algorithms

Chapter 4 presents an analytical analysis of serial CbO based Algorithms. An analytical comparison method can be carried out to compare similar classes of algorithms. This can provide a better insight on how a specific algorithm behaves as opposed to traditional algorithmic analysis based on Big O notation. In addition, a proper theoretical analysis of a series of similar class of algorithms is a complicated task. The empirical results obtained confirm the analytical results obtained. The implementations presented in Chapter 4 were not optimized and were also compiled in debug mode to ensure that the compiled code was an accurate representation of the source code, which in turn directly corresponded to the algorithm.

7.2 Discussion

7.2.1 Debugging Parallel Programs

There was a significant investment in time debugging the parallel implementations of In-Close3. One of the major challenges was the lack proper debugging tools to debug parallel programs compared to serial ones. This is due to the complex and unpredictable way a given parallel program is executed on a parallel computer. Each time you run a parallel program the way workloads are distributed to different cores vary. This is because for scalability reasons a parallel programmer does not explicitly state how different tasks are parallelly allocated to the cores available on a given computer. Instead a runtime scheduler handles the actual job of spawning tasks to available cores.

One of the main lessons learnt during the development of parallel implementations was the importance of doing thorough testing of the serial implementations. A significant number of the bugs discovered in the parallel implementations were found to be minor bugs which manifest in serial implementations only under specific circumstances. The importance of proper software engineering practices such as test-driven development cannot be further understated when one is moving on to writing parallel code.

The most challenging debugging issues came with the MPI implementations. This was because the code needed to be completely rewritten to handle the message passing. It was very easy to introduce errors by not coding the messages that are sent and received. The debugging tools in the Archer supercomputer could tell you where a program crashed but not the reason for it to fail. Since messages are the key components in MPI, unit tests can be developed to see if a message that is sent from one node is received correctly at the receiving end. Data that is sent needs to be serialized (packed) and deserialized (unpacked).

7.2.2 Cross platform Development and Parallel Architectures used for experiments

An early attempt was made to make the codebase that was implemented to be platform independent and compiler independent. Part of the requirement came with the nature of the computers that were to be used for testing the parallel algorithms. Both the Archer Supercomputer and Colfax Clusters used a variety of Intel based Processors running the Linux operating system. The development of the serial and parallel versions of the CbO algorithms used for empirical testing was originally carried out on a Windows Computer and during the latter part of the research on a Mac OS. The compilers used for development were Microsoft Visual C++, Intel C++ Compilers and GNU C++ compilers. For the testing environment the Archer Super Computer which is a Cray XC30 MPP Supercomputer had a Cray C++ Compiler as the preferred compiler.

The implementations were tested on a cluster node containing two Intel Xeon E5-2697v2 with 12 cores each connected through a NUMA configuration. Both Archer and Colfax have these processors. In addition, the implementations were also tested on the older Intel Xeon Phi 7120P coprocessor and the latest Knights Landing Intel Xeon Phi 7120P processor. These processors are optimized for highly parallel workloads and Intel's offering for High Performance Computing. Xeon Phi Processors have a larger core count, typically ranging from 60 to 72 cores per processor and it enables the simultaneous execution of 256 threads. Each core is a simplified Intel X86 processor enabling it to run existing X86 code. However, to get maximum output from the Xeon Phi processors the programs need to be optimized for parallelism.

7.3 Future Work

A Hybrid distributed memory (MPI) and shared memory (OpenMP) implementation can be developed to further improve performance. In this configuration, each physical processor will have one MPI process running while the rest of the cores use the shared memory OpenMP implementation.

The current implementations for the parallel versions of In-Close3 made use of a statically defined large scratchpad for storing concepts. This was originally seen as an optimization to lower the overhead of dynamically allocating memory to store generated and temporary concepts. With the global scratchpad approach for storing generated concepts additional memory needs to be allocated for each core due to the uneven distribution of computational tasks. This can be significant in shared memory parallel implementations as the memory of the computer is shared among all processors in some form. This can be significant for certain datasets where the concepts are generated from specific segments of the recursive call tree. The implementation can be modified to handle dynamic memory allocation and tested with different datasets.

To handle extremely large datasets that cannot be handled at present by one node a combination of the hybrid MPI and OpenMP with dynamic memory allocation can be developed. This will ensure the best performance in terms of memory usage. By parallelizing across multiple physical processors, the overhead of the dynamic memory allocation could be minimized.

The In-Close3 serial algorithm makes use of a bit array to capture the intents that it is computing incrementally. For storing the intents of the generated concepts, a BTree structure is used where integer values of the intent are stored. When In-Close3 is parallelized parts of a given concepts intent is computed by different cores. In the distributed memory parallel implementation, the computed intent resided in multiple memory locations and needs to be assembled by the results process once all computations are done. A quicker way to handle this is to store only the bit array version of the intent as it fully describes the complete intent in the parallel implementation.

The distributed Memory Parallel implementation of In-Close3 can be optimized by logging the performance of each of the specific workers. In the current implementation only one node is assigned to act as the Result Worker, this could act as a bottleneck with the increase of computational nodes. The code can be easily refactored to take into account multiple Result Worker. Another major bottleneck in distributed memory algorithm implementations is the message passing. The master process, worker process and result process should ideally be able send messages with minimum delays. An analysis of a log of timing of messages which are sent through and forth different processes could reveal insights of potential bottlenecks.

References

Alani, H., Kim, S., Millard, D. E., Weal, M. J., Hall, W., Lewis, P. H., & Shadbolt, N. R. (2003). Automatic ontology-based knowledge extraction from web documents. IEEE Intelligent Systems, 18(1), 14–21.

Alessandrini, V. (2015). Shared Memory Application Programming: Concepts and Strategies in Multicore Application Programming. Morgan Kaufmann.

Andrews, S. (2009). In-Close, A fast algorithm for computing formal concepts. In Supplimentary Proceedings of the 17th International Conference on Conceptual Structures.

Andrews, S. (2011). In-Close2, a High Performance Formal Concept Miner. In Proceedings of ICCS 2011, Derby.

Andrews, S. (2014). A partial-closure canonicity test to increase the efficiency of cbo-type algorithms. In International Conference on Conceptual Structures (pp. 37–50).

Andrews, S. (2015). A "Best-of-Breed" approach for designing a fast algorithm for computing fixpoints of Galois Connections. Information Sciences, 295, 633–649. https://doi.org/10.1016/j.ins.2014.10.011

Andrews, S. (2017). Making Use of Empty Intersections to Improve the Performance of CbO-Type Algorithms. In International Conference on Formal Concept Analysis (pp. 56–71).

Andrews, S., & Hirsch, L. (2016). A Tool for Creating and Visualising Formal Concept Trees. In CEUR Workshop Proceedings, 1637 (pp. 1–9).

Anoop, M. (Intel). (2017). Migrate your application to use OpenMP or Intel(R) TBB instead of Intel(R) Cilk(TM) Plus. Retrieved December 1, 2017, from https://software.intel.com/en-us/articles/migrate-your-application-to-use-openmp-or-intelr-tbb-instead-of-intelr-cilktm-plus

Balakirsky, S., & Kramer, T. (2004). Comparing algorithms: Rules of thumb and an example. Retrieved from http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA522948

Barlas, G. (2014). Multicore and GPU Programming: An integrated approach. Elsevier.

Berry, A., Bordat, J.-P., & Sigayret, A. (2007). A local approach to concept generation. Annals of Mathematics and Artificial Intelligence, 49(1–4), 117–136. https://doi.org/10.1007/s10472-007-9063-4

Bordat, J. (1986). Calcul pratique du treillis de Galois d'une correspondance. Mathématiques et Sciences Humaines. Mathematics and Social Sciences, 96, 31–47.

Carpineto, C., & Romano, G. (2004). Concept data analysis theory and applications. Chichester, England; Hoboken, NJ: John Wiley & Sons. https://doi.org/10.1002/0470011297

Chang-Sheng, Z., Jing, R., Hai-Long, H., Long-chang, L., & Bing-ru, Y. (2013). An algorithm on generating lattice based on layered concept lattice. Indonesian Journal of Electrical Engineering and Computer Science, 11(8), 4477–4483.

Chappell, S., & Stokes, A. (2012). Parallel Programming with Intel Parallel Studio XE. John Wiley & Sons.

Chein, M. (1969). Algorithme de recherche des sous-matrices premières d'une matrice. Bulletin Mathématique de La Société Des Sciences Mathématiques de La République Socialiste de Roumanie, 1(13), 21–25.

Cimiano, P. (2009). Ontology Learning Using Corpus-Derived Formal Contexts. In P. Hitzler & H. Schärfe (Eds.), Conceptual structures in Practice (pp. 199–222). (Chapman & Hall.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Introduction to algorithms second edition. The MIT Press.

Couto, L., Oliveira, J. N., Ferreira, M., & Bouwers, E. (2011). Preparing for a literature survey of software architecture using Formal Concept Analysis. In First International Workshop on Model-Driven Software Migration (MDSM 2011) (pp. 64–73).

de Moraes, N. R. M., Dias, S. M., Freitas, H. C., & Zárate, L. E. (2016). Parallelization of the next Closure algorithm for generating the minimum set of implication rules. Artificial Intelligence Research, 5(2), 40.

Desiere, S. (2015). Dell Survey: Structured Data Remains Focal Point Despite Rapidly Changing Information Management | Dell. Retrieved from https://www.dell.com/learn/us/en/vn/press-releases/2015-04-15-dell-survey

Diaz, J., Munoz-Caro, C., & Nino, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. IEEE Transactions on Parallel and Distributed Systems, 23(8), 1369–1386.

Dowling, C. E. (1993). On the Irredundant Generation of Knowledge Spaces. Journal of Mathematical Psychology, 37(1), 49–62. https://doi.org/10.1006/jmps.1993.1003

Flynn, M. J. (1966). Very high-speed computing systems. Proceedings of the IEEE, 54(12), 1901–1909.

Frank, A., & Asuncion, A. (2010). UCI machine learning repository.

Fu, H., & Foghlu, M. O. (2008). A distributed algorithm of density-based subspace frequent closed itemset mining. In High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on (pp. 750–755).

Fu, H., & Nguifo, E. M. (2004). A Parallel Algorithm to Generate Formal Concepts for Large Data. In International Conference on Formal Concept Analysis (ICFCA) (pp. 394–401). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-24651-0_33

Gajdoš, P., & Snášel, V. (2014). A new FCA algorithm enabling analyzing of complex and dynamic data sets. Soft Computing, 18(4), 683–694. https://doi.org/10.1007/s00500-013-1176-6

Ganter, B. (1984). Two basic algorithms in concept analysis. Technische Hochschule Darmstadt Preprint 831.

Ganter, B., & Wille, R. (1999). Concept Lattices of Contexts. In Formal Concept Analysis (pp. 17–61). Springer.

Ganter, B., Stumme, G., & Wille, R. (2002). Formal concept analysis: Methods and applications in computer science. TU Dresden, Http://Www. Aifb. Uni-Karlsruhe. de/WBS/Gst/FBA03. Shtml.

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. ACM SIGPLAN Notices, 42(10), 57–76.

Godin, R., Missaoui, R., & Alaoui, H. (1995). Incremental concept formation algorithms based on Galois (concept) lattices. Computational Intelligence, 11(2), 246–267. https://doi.org/10.1111/j.1467-8640.1995.tb00031.x

Hussain, H., Malik, S. U. R., Hameed, A., Khan, S. U., Bickler, G., Min-Allah, N., … others. (2013). A survey on resource allocation in high performance distributed computing systems. Parallel Computing, 39(11), 709–736.

Jeffers, J., & Reinders, J. (2015). High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches. Morgan Kaufmann.

Jeffers, J., Reinders, J., & Sodani, A. (2016). Intel Xeon Phi Processor High Performance Programming, 2nd Edition. Morgan Kaufmann.

Kaytoue, M., Kuznetsov, S. O., & Napoli, A. (2011). Biclustering numerical data in formal concept analysis. In International Conference on Formal Concept Analysis (pp. 135–150).

Kedia, K. (2009). Hybrid Programming with OpenMP and MPI.

Kengue, J. F. D., Valtchev, P., & Djamegni, C. T. (2007). Parallel computation of closed itemsets and implication rule bases. In International Symposium on Parallel and Distributed Processing and Applications (pp. 359–370).

Keutzer, K., & Mattson, T. (n.d.). Introduction to Design Patterns for Parallel Computing.

Krajca, P., & Vychodil, V. (2009). Distributed algorithm for computing formal concepts using map-reduce framework. In International Symposium on Intelligent Data Analysis (pp. 333–344).

Krajca, P., Outrata, J., & Vychodil, V. (2008). Parallel recursive algorithm for FCA. In CLA (Vol. 2008, pp. 71–82).

Krajca, P., Outrata, J., & Vychodil, V. (2010). Advances in algorithms based on CbO. CEUR Workshop Proceedings, 672, 325–337.

Krajca, P., Outrata, J., & Vychodil, V. (2010a). Advances in algorithms based on CbO.

Krajca, P., Outrata, J., & Vychodil, V. (2010b). Parallel algorithm for computing fixpoints of Galois connections. Annals of Mathematics and Artificial Intelligence, 59(2), 257–272.

Krötzsch, M., & Ganter, B. (2009). A brief introduction to formal concept analysis. In P. Hitzler & H. Schärfe (Eds.), In Conceptual Structures in Practice - Studies in Informatics Series (pp. 3–16). Chapman and Hall/CRC.

Kruskal, W. H., & Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. Journal of the American Statistical Association, 47(260), 583–621.

Kumary, R. (2005). Research Methodology: A Step Guide for Beginners, 2nd edition. Pearson.

Kuznetsov, S. O., & Obiedkov, S. a. (2002). Comparing performance of algorithms for generating concept lattices. Journal of Experimental & Theoretical Artificial Intelligence, 14(2–3), 189–216. https://doi.org/10.1080/09528130210164170

Langmead, B., & Nellore, A. (2018). Cloud computing for genomic data analysis and collaboration. Nature Reviews Genetics, 19(4), 208.

Lilja, D. J. (2005). Measuring computer performance: a practitioner's guide. Cambridge university press.

Lindig, C. (2000). Fast concept analysis. In Working with Conceptual Structures-Contributions to ICCS 2000 (pp. 152–161). Shaker Verlang. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.143.948&rep=rep1&type=pdf

Mani, K., & Jee, B. (2007). On the Edge: A Comprehensive Guide to Blade Server Technology.

Mattson, T. G., Sanders, B., & Massingill, B. (2004). Patterns for parallel programming. Pearson Education.

Norris, E. (1978). An algorithm for computing the maximal rectangles of a Binary Relation. Rumanian Revue of Pure and Applied Mathematics 2, 23 SRC-, 243–250.

Nourine, L., & Raynaud, O. (1999). A fast algorithm for building lattices. Information Processing Letters, 71(5–6), 199–204. https://doi.org/10.1016/S0020-0190(99)00108-8

Old, J., & Priss, U. (2004). Some open problems in formal concept analysis. Journal Of Universal Computer Science, 3874(8), 306–308.

Outrata, J. (2015). Computing and Applying Formal Concepts - Algorithms and Methods. Palacký University Olomouc.

Outrata, J., & Vychodil, V. (2012). Fast algorithm for computing fixpoints of Galois connections induced by object-attribute relational data. Information Sciences, 185(1), 114–127. https://doi.org/10.1016/j.ins.2011.09.023
Pacheco, P. (2011). An introduction to parallel programming. Elsevier.

Pacheco, P. S. (1997). Parallel programming with MPI. Morgan Kaufmann.

Poelmans, J., Ignatov, D. I., Kuznetsov, S. O., & Dedene, G. (2013a). Formal concept analysis in knowledge processing: A survey on applications. Expert Systems with Applications, 40(16), 6538–6560. https://doi.org/10.1016/j.eswa.2013.05.009

Prasad, S. K., Gupta, A., Rosenberg, A. L., Sussman, A., & Weems, C. C. (2015). Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses. Morgan Kaufmann.

Priss, U. (2006). Formal concept analysis in information science. Arist, 40(1), 521–543.

Reyes-Ortiz, J. L., Oneto, L., & Anguita, D. (2015). Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. Procedia Computer Science, 53, 121–130.

Ross, S. M., Morrison, G. R., & Mahwah, N. J. (2004). Experimental Research Methods.

Saecker, M., & Markl, V. (2013). Big data analytics on modern hardware architectures: A technology survey. In Business Intelligence (pp. 125–149). Springer.

Sarmah, A. K., Hazarika, S. M., & Sinha, S. K. (2015). Formal concept analysis: current trends and directions. Artificial Intelligence Review, 44(1), 47–86.

Silva, L. M., & Buyya, R. (1999). Parallel programming models and paradigms. High Performance Cluster Computing: Architectures and Systems, 2, 4–27.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., & Dongarra, J. (1998). MPI: The Complete Reference (Vol. 1). The MIT Press Cambridge, MA.

Strohmaier, E., Dongarra, J., Simon, H., & Meur, M. (2018). Top 500 - June 2018.

Strok, F., & Neznanov, A. (2010). Comparing and analyzing the computational complexity of FCA algorithms. Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on - SAICSIT '10, 417–420. https://doi.org/10.1145/1899503.1899557

Stumme, G., Taouil, R., Bastide, Y., Pasquier, N., & Lakhal, L. (2000). Fast Computation of Concept lattices Using Data Mining Techniques. In KRDB (pp. 129–139).

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 30(3), 202–210.

Sutter, H., & Larus, J. (2005). Software and the concurrency revolution. Queue, 3(7), 54–62.

Tilley, T., Cole, R., Becker, P., & Eklund, P. (2005). A Survey of Formal Concept Analysis Support for Software Engineering Activities. Pp Berlin Heidelberg Springer, 250–271. https://doi.org/citeulike-article-id:257415

Turner, A., & McIntosh-Smith, S. (2017). A survey of application memory usage on a national supercomputer: an analysis of memory requirements on ARCHER.

Uschold, M., & Gruninger, M. (1996). Ontologies: Principles, methods and applications. The Knowledge Engineering Review, 11(2), 93–136.

Vladimirov, A., & Karpusenko, V. (2013). Parallel Programming and Optimization with Intel Xeon Phi Coprocessors. ColeFax International, (May), 520.

Vladimirov, A., Asai, R., & Karpusenko, V. (2015). Parallel Programming and Optimization with Intel Xeon Phi Coprocessors: Handbook on the Development and Optimization of Parallel Applications for Intel Xeon Processors and Intel Xeon Phi Coprocessors. Colfax International.

Wall, M. (2014). Big Data: Are you ready for blast-off? - BBC News. Retrieved December 1, 2016, from http://www.bbc.com/news/business-26383058

Wille, R. (1982). Restructuring lattice theory: an approach based on hierarchies of concepts. In Ordered sets (pp. 445–470). Springer.

Wille, R. (2005a). Formal concept analysis as mathematical theory of concepts and concept hierarchies. In Formal concept analysis (pp. 1–33). Springer.

Wille, R. (2005b). Formal Concept Analysis as Mathematical Theory of Concepts and Concept Hierarchies. Formal Concept Analysis, 1–33. https://doi.org/10.1007/11528784_1

Wray, T., & Eklund, P. (2011). Concepts and collections: A case study using objects from the brooklyn museum. CEUR Workshop Proceedings, 801, 109–120.

Wray, T., & Eklund, P. (2014). Using Formal Concept Analysis to Create Pathways through Museum Collections. Proceedings of the 3rd International Workshop "What Can Fca Do for Artificial Intelligence"? (Fca4ai 2014), 9–16.

Xu, B., de Fréin, R., Robson, E., & Foghlú, M. Ó. (2012). Distributed formal concept analysis algorithms based on an iterative mapreduce framework. In International Conference on Formal Concept Analysis (pp. 292–308).

Yang, C.-T., Huang, C.-L., & Lin, C.-F. (2011). Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. Computer Physics Communications, 182(1), 266–269.

Appendix – A – Serial Algorithm Implementations

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize,
                         VECTOR_TYPE *BParentBit, int y) {
   int C[OBJECTSIZE];
   short int D[ATTRIBUTESIZE];
   int CSize, DSize;
   VECTOR_TYPE BChildBit[VECTOR_MAX_COLS_CELLS]; //the current intent in Boolean form
   memcpy(BChildBit,BParentBit,nArray*VECTOR_SIZE_BYTES);
   TQueue q;
   int j

   for (j=y; j<n; j++) {
      if (! isMember(j,BChildBit)) {
         aIntersectionColj(C,CSize,A,ASize,j);
         if (isEqual(A, ASize, C,  CSize)) {
            insert(BChildBit,B,BSize,j);
         }
         else {
            if (buptoJisEqualtoPartialClosureOfCuptoJBit2(BChildBit,C,CSize,j)) {
               put(q,C,CSize,j);
            }
         }
      }
   }
   // Print Concept
   conceptno++;
   int concept = conceptno;
   while (get(q,C,CSize,j)) {
      copyInsert(BChildBit, D, DSize, B, BSize, j);
      ComputeConceptsFrom(C,CSize, D, DSize, BChildBit, j+1);
   }
}
```

Figure A.1, Listing of CbO-PC-DBF program code

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize,
                         VECTOR_TYPE *BParentBit, int y, VECTOR_TYPE *NBit[]) {
   int C[OBJECTSIZE];
   short int D[ATTRIBUTESIZE];
   int CSize,  DSize;
   VECTOR_TYPE *DBit;
   int concept = conceptno;
   VECTOR_TYPE BChildBit[VECTOR_MAX_COLS_CELLS]; //the current intent in Boolean form
   memcpy(BChildBit,BParentBit,nArray*VECTOR_SIZE_BYTES);
   TQueue q;
   VECTOR_TYPE *MBit[ATTRIBUTESIZE];
   int j;
   for (j=y; j<n; j++) {
      copyRowArrMN(&MBit[j],NBit[j]);
      if (! isMember(j,BChildBit)) {
         if (isNjSubSetofBuptoJBit(NBit[j],BChildBit,j)) {
            aIntersectionColj(C,CSize,A,ASize,j);
            if (isEqual(A, ASize, C,  CSize))  {
               insert(BChildBit,B,BSize,j);
            }
            else {
               if (buptoJisEqualtoPartialClosureOfCuptoJBit2(BChildBit,C,CSize,j)) {
                  copyInsert(BChildBit, D, DSize, B, BSize, j);
                  ComputeConceptsFrom(C,CSize,D, DSize, BChildBit, j+1, MBit);
               }
               else {
                  copyRowArrMPartialClosure (&MBit[j], j, C,CSize);
               }
            }
         }
      }
   }
   // Print Concept
   conceptno++;
   concept = conceptno; // keeps track of the current concept
}
```

Figure A.2, Listing of CbO-PC-ICF-DF program code

.

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize,
                       VECTOR_TYPE *BParentBit, int y, VECTOR_TYPE *NBit[]) {
    int C[OBJECTSIZE];;
    short int D[ATTRIBUTESIZE];
    int CSize,  DSize;
    VECTOR_TYPE *DBit;
    int concept = conceptno;
    VECTOR_TYPE BChildBit[VECTOR_MAX_COLS_CELLS]; //the current intent in Boolean form
    memcpy(BChildBit,BParentBit,nArray*VECTOR_SIZE_BYTES);
    TQueue q;
    VECTOR_TYPE *MBit[ATTRIBUTESIZE];
    int j;
    for (j=y; j<n; j++) {
       copyRowArrMN(&MBit[j],NBit[j]);
       if (! isMember(j,BChildBit)) {
          if (isNjSubSetofBuptoJBit(NBit[j],BChildBit,j)) {
             aIntersectionColj(C,CSize,A,ASize,j);
            if (isEqual(A, ASize, C,  CSize))  {
               insert(BChildBit,B,BSize,j);
            }
            else {
               if (buptoJisEqualtoPartialClosureOfCuptoJBit2(BChildBit,C,CSize,j)) {
                  put(q,C,CSize,j)
               }
               else {
                  copyRowArrMPartialClosure (&MBit[j],j, C,CSize);
               }
            }
         }
      }
   }
   // Print Concept
   conceptno++;
   concept = conceptno; // keeps track of the current concept
   while (get(q,C,CSize, j)) {
      copyInsert(BChildBit, D, DSize, B, BSize, j);
      ComputeConceptsFrom(C,CSize,D, DSize, BChildBit, j+1, MBit);
   }
}
```

Figure A.3, Listing of CbO-FC-ICF-DBF program code

```
void ComputeConceptsFrom(int *A, int ASize, short int *B, int BSize,
                       VECTOR_TYPE *BParentBit, int y, VECTOR_TYPE *NBit[]) {
    int C[OBJECTSIZE];;
    short int D[ATTRIBUTESIZE];
    int CSize,  DSize;
    VECTOR_TYPE *DBit;
    int concept = conceptno;
    VECTOR_TYPE BChildBit[VECTOR_MAX_COLS_CELLS]; //the current intent in Boolean form
    memcpy(BChildBit,BParentBit,nArray*VECTOR_SIZE_BYTES);
    TQueue q;
    VECTOR_TYPE *MBit[ATTRIBUTESIZE];
    int j;
    for (j=y; j<n; j++) {
       copyRowArrMN(&MBit[j],NBit[j]);
       if (! isMember(j,BChildBit)) {
          if (isNjSubSetofBuptoJBit(NBit[j],BChildBit,j)) {
            aIntersectionColj(C,CSize,A,ASize,j);
            if (isEqual(A, ASize, C,  CSize))  {
               insert(BChildBit,B,BSize,j);
            }
            else {
               if (buptoJisEqualtoPartialClosureOfCuptoJBit2(BChildBit,C,CSize,j)) {
                  put(q,C,CSize,j)
               }
               else {
                  copyRowArrMPartialClosure (&MBit[j],j, C,CSize);
               }
            }
         }
      }
   }
   // Print Concept
   conceptno++;
   concept = conceptno; // keeps track of the current concept
   while (get(q,C,CSize, j)) {
      copyInsert(BChildBit, D, DSize, B, BSize, j);
      ComputeConceptsFrom(C,CSize,D, DSize, BChildBit, j+1, MBit);
   }
}
```

Figure A.4, Listing of CbO-PC-ICF-DBF program code

Appendix – B – Distributed Memory Algorithm Implementations

```
 1 void MPICreateDataType() {
 2         int const max = 9;
 3         int array_of_blocklengths[max];
 4         MPI_Aint array_of_displacements[max];
 5 
 6         MPI_Datatype array_of_types[max];
 7         array_of_blocklengths[0] = OBJECTSIZE;
 8         array_of_types[0] = MPI_INT;
 9         array_of_blocklengths[1] = 1;
10         array_of_types[1] = MPI_INT;
11         array_of_blocklengths[2] = VECTOR_MAX_COLS_CELLS;
12         array_of_types[2] = MPI_LONG_LONG_INT;
13         array_of_blocklengths[3] = 1;
14         array_of_types[3] = MPI_INT;
15         array_of_blocklengths[4] = 1;
16         array_of_types[4] = MPI_INT;
17         array_of_blocklengths[5] = 1;
18         array_of_types[5] = MPI_SHORT;
19         array_of_blocklengths[6] = 1;
20         array_of_types[6] = MPI_INT;
21         array_of_blocklengths[7] = VECTOR_MAX_COLS_CELLS;
22         array_of_types[7] = MPI_LONG_LONG_INT;
23         array_of_blocklengths[8] = (VECTOR_MAX_COLS_CELLS) * ATTRIBUTESIZE;
24 
25         array_of_types[8] = MPI_LONG_LONG_INT;
26 
27         tParaCompound msg;
28         MPI_Aint addr[max];
29         MPI_Get_address(&msg, &addr[0]);
30         MPI_Get_address(&msg.ASize, &addr[1]);
31         MPI_Get_address(&msg.BParentBit, &addr[2]);
32         MPI_Get_address(&msg.y, &addr[3]);
33         MPI_Get_address(&msg.c, &addr[4]);
34         MPI_Get_address(&msg.level, &addr[5]);
35         MPI_Get_address(&msg.flag, &addr[6]);
36         MPI_Get_address(&msg.NBitFlag, &addr[7]);
37         MPI_Get_address(&msg.NBit, &addr[8]);
38 
39         array_of_displacements[0] = 0;
40         for (int r = 1; r < max; r++)
41                 array_of_displacements[r] = addr[r] - addr[0];
42 
43         MPI_Type_create_struct(max, array_of_blocklengths, array_of_displacements,
44                 array_of_types, &TaskType);
45         MPI_Type_commit(&TaskType);
46         // Declare memory for results Buffer
47         maxResultsBufferSize = maxSizeOfVars + maxLinearSize;
48         resultsBuffer = new char[maxResultsBufferSize];
49 }
```

Figure B.1, MPI CreateDataType() function

On the MPI side of things the message structure that is used to capture the C structure needs to be created. The `MPI_Data` type allows the definition of both simple and structure type variables which can be serialized. The `MPI_Type_create_struct()` function shown in line 43 (See *Figure B.1*) allows to create a user defined MPI data structure which is the equivalent of a C Structure. To do this the individual members of the structure needs to be defined, this is achieved in lines 7 to 22 where each member is defined as an array of `MPI_Datatype` called `array_of_types`. The size of each of the data types also needs to be defined MPI and the integer array defined in line 3 is used for this purpose. The `MPI_Get_address` is a function that can be used to find the memory locations of each member of a C structure. Lines 29 to 37 capture the memory locations of all the members of the C structure `tParaCompound`. Lines 39 to 42 are used to calculate the offset of each of the structure members.

Finally in line 43 and 45 a MPI Data type that represents a structure is created using the `MPI_Type_create_struct()` and `MPI_Type_commit()` MPI functions. Creation of a structure type in MPI is a one time process and this needs to be executed in all nodes. MPI function in line 43 requires the number of items in the structure (`max`), the size of each element, the member (`array_of_displacements`) contains the offset of each member from the beginning of the structure (`array_of_displacements`), the data types of each member (`array_of_types`) and the `MPI_DataType` variable which will be defined (`TaskType`).

```
1     // Used by the Master Process pack the Task details to be spawned, this
2     //   is later sent to a Worker Process who is ready
3     void MPIPackData(tParaCompound &msg, const int *AV, const int &ASize, const VECTOR_TYPE *BParentBit,
4     const int &y, const int &c,  VECTOR_TYPE *NBit[], const short int &level) {
5             memcpy(msg.A, Av,ASize*sizeof(int));
6             msg.ASize = ASize;
7             memcpy(msg.BParentBit, BParentBit, nArray*sizeof(VECTOR_TYPE));
8             msg.y = y;
9             msg.c = c;
10            int count = 0;
11            // Set NBitFlag to zero
12
13            for (int k = 0; k < VECTOR_MAX_COLS_CELLS; k++)
14                    msg.NBitFlag[k] = 0;
15            VECTOR_TYPE NBitFlag[VECTOR_MAX_COLS_CELLS] = { 0 };
16            for (int r = 0; r < MAXATTRIBUTES; r++) {
17                    if ((NBit[r]) == NULL) {
18                            ClearBit1(NBitFlag, r);
19                    }
20                    else {
21                            SetBit1(NBitFlag, r);
22                            memcpy(&msg.NBit[count*nArray], NBit[r], nArray * sizeof(VECTOR_TYPE));
23                            count++;
24                    }
25            }
26            memcpy(msg.NBitFlag, NBitFlag, (VECTOR_MAX_COLS_CELLS)*sizeof(VECTOR_TYPE));
27            msg.level = level;
28            msg.flag = true;  // Valid Data
29    }
```

Figure B.2, MPIPackData() function

To optimize message passing a `NBitFlag` variable was introduced (See *Figure B.2*), it keeps track of which `NBit[]` array elements contains values and which contain NULL values. The `memcpy()` function in line 22 is carried out only for entries that have values. This enables packing of `NBit[]` elements when serialization and deserialization takes place.