Memory Management and Parallelization of Data Intensive All-to-all Comparison in Shared-memory Systems

by

Anaththa Pathiranage Dhanushka Krishnajith,

B.Sc. Eng (Hons)

PhD Thesis

Submitted in Fulfilment

of the Requirements

for the Degree of

Doctor of Philosophy

at the

Queensland University of Technology

Faculty of Science and Engineering

October 2014

Keywords

All-to-all comparison, bioinformatics, composition vector method, memory management, multi-core computers, paging algorithms, parallelization, shared-memory systems

Abstract

Parallelization of Data Intensive All-to-all Comparison (DIAC) in limited memory is challenging and widely found in many scientific and generic applications. In an all-to-all comparison, every data item in a group of data items is pair-wise compared with the rest of the data items in the group. A large number of data items in a data set is emphasised by the term "data intensive" in DIAC. In the class of DIAC targeted in this research, the data items need to reside in memory to complete comparisons and the process of loading data items to memory is significantly time consuming. Due to the large number of data items, they may not all fit completely into memory. Thus, data items generally need to be paged in and out more than once to complete a DIAC. Therefore, increasing the speed of a DIAC in limited memory requires overcoming two challenging problems. Firstly, the number of repeated loads of data items must be minimized leading to an challenging problem to solve due to the large combinatorial number of choices over the life-span of a DIAC calculation. Secondly, the amount of comparisons completed in parallel must be maximized. This is an even more challenging problem because the number of parallel comparisons is limited by the available memory capacity and the rate of bringing comparisons to memory is constrained by the speed of the loads.

To solve the above mentioned two challenging problems, this research develops a systematic approach to minimize the runtime of memory-constrained DIAC for uni-processor and shared-memory multi-processor platforms through efficient utilization

of available computing resources. The three main contributions of the thesis include: first, a novel memory management (paging) algorithm is proposed for faster and scalable computing of DIAC through minimizing the loads of data items to memory. The performance improvement of the approach over previous work ranges from 7% to 31.9 times improvement in speed and is demonstrated through benchmark examples. The parameter settings for the proposed algorithm are derived theoretically and a theoretical close-fitting lower bound is also established for the minimum number of loads required to complete a DIAC. Second, novel parallelization technique is introduced. Incorporating dynamic scheduling with static scheduling, it utilizes a problem specific pattern developed based on the prior knowledge of the targeted problem abstraction. This technique significantly minimizes the scheduling overhead of parallelizing complex problems such as DIAC while still producing quality schedules. Implementing this technique, a novel parallelization algorithm is presented which dramatically increases the speed of bioinformatics DIAC computing in comparison with their sequential counterparts while efficiently managing the memory. In an octa-core system, a test of our algorithm for a benchmark example shows a runtime of two weeks reduced to less than two days. Furthermore, the parameter settings of the algorithm are theoretically derived for scenarios commonly found in general computing platforms. The third contribution of this thesis is our development of a novel simulator and a parallel framework for DIAC to assist evaluations of parallel algorithms including the one proposed in this thesis. The simulation environment is fast and lightweight for simulation of shared-memory multi-processor systems and exploits the advantage of the assumed uninterrupted and continuous processor-task association of the proposed parallelization algorithm. It can be used to simulate other parallelization algorithms with similar properties as well. The framework facilitates rapid implementations of parallel algorithms for DIAC. It clearly separates the decision making logic form implementation details.

Contents

Abstract **i**

Nomenclature **xi**

Acronyms and Abbreviations **xv**

Certification of Thesis **xvii**

Acknowledgments **xix**

Chapter 1 Introduction **1**

1.1 Research Background 3

1.2 Motivations and Significance 6

1.3 Statement of the Research Problem 8

1.4 Main Contributions 8

1.5 Thesis Outline 10

1.6 Scope and Limitations of the Thesis 11

1.7 Publications During PhD Research 12

Chapter 2 Literature Review **13**

2.1 All-to-All Comparison 14

2.1.1 All-to-All Comparison in Composition Vector Method 16

2.1.2 Other All-to-All Comparison Problems 19

2.1.3 All-to-All Comparison in Bioinformatics Applications 20

2.2 Memory Management in Single-core Computers 22

2.3 Specific Memory Improvements for the CV Method 25

2.4 Parallelization of All-to-All Comparison 26

2.4.1 Parallelization Specific to All-to-All Comparison 27

2.4.2 Parallelization of Tasks Sharing Data 30

2.4.3 Summary of Parallelization of All-to-All Comparison 32

2.5 Analysis of Popular Parallelization Strategies 33
2.6 Popular Parallel Computing Platforms 36
2.7 Simulators for Parallel Algorithms 41
2.8 Summary of the Literature Review 43
Chapter 3 Memory Management in Uni-processor Systems 45
3.1 Problem Formalization and Strategy Formulation 46
3.2 Algorithm Development 48
3.2.1 Algorithm 48
3.2.2 Graphical Illustration of the Algorithm 51
3.2.3 Illustrative Example of the Algorithm 51
3.2.4 Discussion on Algorithm 4 54
3.3 Theoretical Results 54
3.3.1 Theorems for Algorithm 4 55
3.3.2 Lower bound of Required Loads for All-to-All Comparison . . 63
3.3.3 Time Complexity of Algorithm 4 67
3.4 Scalable Memory Management Algorithm 68
3.5 Experimental Validation 72
3.5.1 Benchmark Examples 72
3.5.2 Experimental Settings 73
3.5.3 Experimental Design 73
3.5.4 Virtual Memory 74
3.5.5 Performance of the Algorithm 75
3.5.6 Comparison with Generic I/O Optimization Algorithms 77
3.5.7 Effect of Sorting of Composition Vectors 78
3.5.8 Performance with Different Memory Sizes 79
3.5.9 Data Structure Used for Composition Vectors 80
3.5.10 Avoiding Virtual Memory 81
3.5.11 Summary of the Experimental Results 82
3.6 Summary of the Chapter 82
Chapter 4 Preparation to Solve the DIAC Parallelization Problem 84

4.1 Modelling the DIAC Parallelization Problem 85
4.1.1 Modelling as a Scheduling Problem 86
4.1.2 Self-Adjusting Dynamic Scheduling (SADS) 90
4.1.3 Directed Acyclic Graphs (DAG) 98
4.1.4 Heuristic Functions Based Scheduling 99
4.1.5 Using Local Search with Traditional Memory Management Techniques 110
4.2 Upper bound for the Parallel Performance Gain 115
4.3 Summary of the Chapter 119
Chapter 5 Parallelization and Memory Management in Multi-core Systems 121
5.1 Development of Scheduling Strategies 122
5.1.1 Aspect 1: Maximizing Comparisons and Minimizing Loads . . 123
5.1.2 Aspect 2: Using the Knowledge of DIAC Problem 125
5.1.3 Aspect 3: Minimizing Synchronization Overhead 126
5.1.4 Aspect 4: Time Spent on Scheduling 127
5.2 Algorithm Design 128
5.2.1 Designing the Pattern 129
5.2.2 Algorithm Development 136
5.3 Analysis of the Algorithm Behaviour 142
5.3.1 Examples for Understanding the Behaviour of the Algorithm . 143
5.3.2 Factors Affecting the Behaviour of the Algorithm 148
5.4 Determination of Algorithm Parameters 150
5.4.1 Maximum Size of V_{im} (Q) in Algorithm 7 152
5.4.2 Optimum Size of V_{im} (Q) in Algorithm 7 155
5.4.3 Determination of Optimum Q by using Simulations 162
5.4.4 Validating Theoretical Results 163
5.5 Summary of the Chapter 166
Chapter 6 Algorithm Implementation and Experiments 167
6.1 Extending Algorithm 7 to Solve Practical Problems 168
6.1.1 Handling Non-uniform Sizes and Variable Q 168

6.1.2 Unpredictable Load and Comparison Times 172
6.2 Algorithm Implementation 174
6.3 Preprocessing Data Items in Bioinformatics CV method 176
6.4 Experimental Results 179
6.4.1 Behaviour of Algorithm 8 180
6.4.2 Performance of the Proposed Algorithm 8 184
6.5 Summary of the Chapter 188
Chapter 7 Simulators, Visualizers and Frameworks **191**
7.1 The Simulator 192
7.1.1 Simulator Design 194
7.2 The Parallelization Framework 199
7.3 Visualizers 204
7.4 Summary 206
Chapter 8 Conclusion **207**
8.1 Major Contributions 210
8.2 Minor and Incremental Contributions 212
8.3 Limitation and Future Work 214
Appendix A Estimating Load Times and Composition Vector's Size **215**
Bibliography **223**

List of Tables

2.1 Similarities between the problem addressed by Wu et al. [2009] versus our DIAC problem 23
2.2 Differences between the problem addressed by Wu et al. [2009] versus our DIAC problem 23

3.1 Comparison between virtual memory managed by the operating system and program itself 75
3.2 Analysing performance of Algorithm 5 with the program by Yu et al. [2010a] 76
3.3 Comparing Algorithm 5 with CVTree program's algorithm 77
3.4 Comparing Algorithm 5 with LRU I/O optimization algorithm 78
3.5 Effect of ordering genomic sequences in Algorithm 5 79

4.1 Sample sequences of operations with their validity according to the model 90
4.2 Heuristic functions used for the decision of starting a *load*. The heuristics to decide which data item must be loaded is discussed in Table 4.4 103
4.3 Heuristic functions used for the decision of starting a comparison . . . 104
4.4 Heuristic functions used for the decision of what to load 105
4.5 Heuristic functions used for the decision of what to compare 106
4.6 Heuristic functions used for the decision of unloading data items . . . 107
4.7 Runtime comparison of different local search based approaches 115

6.1 Comparison between average time spent for generating a *composition vector* from scratch and time spent for reading it from the disk 186
6.2 The properties of the computers which are used for the experiments . 186
6.3 The list of data sets used in the experiments and their properties 187
6.4 A comparison between our sequential memory management algorithm (Algorithm 5) and our parallel algorithm (Algorithm 8). 188

A.1 Notations used in Table A.2. 216
A.2 Steps in calculating a *composition vector* and complexity analyses of each step. Please see Table A.1 for the notations. 217

List of Figures

1.1 A sample correlation matrix 3

2.1 Categories of *bag-of-tasks* problems 30
2.2 The taxonomy of computer architecture 38
2.3 Memory models in parallel computers 39

3.1 An example of the behaviour of Algorithm 4 53
3.2 Behaviour of the number of loads required versus β in case I 60
3.3 Behaviour of the number of loads required versus β in case II 62
3.4 Comparison of the number of *loads* required between Algorithm 4 and the theoretical lower-bound 66
3.5 Behaviour of the execution time of Algorithm 5 versus the available memory capacity 80
3.6 The data structure developed and used in the applications for CV method 81

4.1 A sample search tree for a DIAC in SADS 95
4.2 Direct Acyclic Graph (DAG) representation of DIAC parallelization problem 99
4.3 Procedure for making decisions based on the heuristic functions . . . 102
4.4 Comparison of decision making based on heuristic function to Algorithm 7 in different settings 109
4.5 Model for producing schedules based on the order of a supplied sequence of comparisons. 112

4.6 Comparison between the speed-up of Algorithm 8 and the theoretical upper bound 118

5.1 How correlation matrix is divided in Algorithm 5 130
5.2 Behaviour of the simple parallel algorithm (Algorithm 6) 133
5.3 Behaviour of the pipe-lined parallel algorithm 135
5.4 Separation of rows into two sets in Algorithm 7 136
5.5 Behaviour of Algorithm 7 with $Q = 1$ 144
5.6 Behaviour of Algorithm 7 with various values for Q 146
5.7 Behaviour of Algorithm 7 with $Q = 3$ 147
5.8 A scenario where $Q = 1$ may be better in Algorithm 7 149
5.9 The behaviour of Algorithm 7 with two parallel *loads* are allowed . . 151
5.10 Behaviour of Algorithm 7 in Scenario 1 157
5.11 Behaviour of Algorithm 7 in Scenario 2 159
5.12 Validation of theoretical results to calculate the optimum Q for Algorithm 7 164
5.13 Behaviour of the optimum Q value for Algorithm 7 against various settings 165

6.1 Behaviour of Algorithm 7 in real computers 173
6.2 Speed-up of Algorithm 8 with load to comparison time ratio 182
6.3 Speed-up of Algorithm 8 against the memory capacity 183
6.4 Speed-up of Algorithm 8 against the number of processors 185
6.5 Visualizing the performance of Algorithm 7 189

7.1 A depiction of how the simulator executes a code and simulates long running tasks in between 196
7.2 Class hierarchy of the simulator implementation 198
7.3 The *Rules* depicted as a black-box for decision making 201
7.4 The flowchart of the framework which uses *Rules* as the decision maker. 203

7.5 A screen-shot of the visualizer tool 205

A.1 Significant difference in time of nearly similar sized genomes in T Vector calculation 219

A.2 Profiling of T-Vector generation step of two nearly similar sized gene sequences 220

A.3 Variation of *load* times when *composition vectors* are loaded from the disk in Data Set 2 221

Nomenclature

α	Size of set A
β	Size of set B
A	A set of data items used in algorithms
A'	A set of data items used in algorithms
A''	A set of data items used in algorithms
B	A set of data items used in algorithms
B'	A set of data items used in algorithms
B''	A set of data items used in algorithms
C	Correlation matrix
$C[x, y]$	Value of correlation matrix at column x and row y; $0 \leq x, y < N$
$C_{i,j}$	Operation of comparison between the data items G_i and G_j; $0 \leq i, j < N$
G_i	Data item of a data set; $0 \leq i < N$
h_i	Different heuristic functions used to solve DIAC parallelization; $1 \leq i \leq 4$
K	A parameter in the CV method [Yu et al., 2010a]

L Number of *loads* in an algorithm

L_P Maximum number of efficient parallel *loads* in a system

L_i Operation of *load* of G_i; $0 \leq i < N$

L_{tmin} The minimum number of *loads* required to complete an *all-to-all comparison* according to Theorem 8

M The maximum number of data items which can be held in memory due to memory capacity limitations

N The number of data items in a data set

Q The maximum size of V_{im}

Q_{max} Maximum size of Q

Q_{opt} Optimum size of Q

R_D A disk resource

R_M A byte of memory resource

$R_M(G_i)$ The amount of R_M required to *load* G_i and keep it in memory; $0 \leq i < N$

R_P A processor resource

S Speed up of a parallel algorithm compared to its sequential counterpart

S_{max} The upper bound of the maximum speed gain of an *all-to-all comparison* by using parallel execution in a shared-memory computer capable only of sequential loads compared to the best possible sequential *all-to-all comparison* algorithm

t_c Average time required for a comparison

t_l Average time required for a *load*

t_{pmin} The lower-bound of the time required to complete an *all-to-all comparison* in a shared-memory computer which is only capable of sequential *loads*

t_{smin} The minimum time required to complete an *all-to-all comparison* without parallelism

U A set of data items used in algorithms

U_i Operation of unloading G_i; $0 \leq i < N$

V A set of data items used in algorithms

V_i preprocessed data item of G_i; $0 \leq i < N$

V_{im} The data items of set V which are loaded to memory (im for In Memory)

β_{min} Value of β to get L_{min}

$\overline{L}$ Upper-bound of the number of *loads* in an algorithm

$\underline{L}$ Lower-bound of the number of *loads* in an algorithm

$\overline{L_{min}}$ Upper-bound of L_{min}

$\underline{L_{min}}$ Lower-bound of L_{min}

L_{min} Minimum of L

Acronyms and Abbreviations

CV	Composition Vector
DAG	Directed Acyclic Graphs
DIAC	Data Intensive All-to-all Comparison
FA	Future Aware
FIFO	First In First Out
GS	Genomic Sequence
GSes	Genomic Sequences
HPC	High Performance Computing
LAN	Local Aria Network
MIMD	Multiple Instructions Multiple Data
SADS	Self Adjusting Dynamic Scheduling
SIMD	Single Instruction Multiple Data
LFU	Least Frequently Used
LRU	Least Recently Used

Certification of Thesis

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher educational institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

QUT Verified Signature

Signed:

Date: 19/11/2014

Acknowledgments

Work on this journey, while interesting, was not easy and would not have been possible without support and guidance from a lot of people along the way. First and foremost, I would like to express my utmost gratitude to my mother, father and wife who have been, and still are, my biggest supporters. My wife, Gawri Edussuriya who has not only heard but also felt all ups and downs in this journey, was one of the biggest driving force behind me throughout. Their never ending support, as well as their guidance, comfort, compassion and perspective have allowed me to achieve more than I could ever imagine. I am eternally grateful to them for everything they have done for me and they will never know how much of a positive influence they have been on my life.

I would like to express my warm and sincere gratitude to Professor Yu-chu (Glen) Tian, who is the head of the Networks and Communications discipline at Queensland University of Technology (QUT), for all the support and guidance he has given me in order to overcome numerous obstacles in this journey. I am really grateful to him for providing an excellent work environment around me and for the opportunities he has made for me, in working with world-class scientists and attending international conferences. I would like to thank him for his effort, patience and excellent guidance during my PhD journey.

I wish to further extend my sincere thanks to my associate supervisors, Dr Wayne Kelly and Dr Ross Hayward, for their valuable support and advice throughout this journey.

I would also like to thank Professor Zu-Guo Yu for providing valuable support and guidance specially related to bioinformatics applications. Also, I wish to convey my gratitude to QUT for providing me this opportunity and also to all the administrative staff in HDR research coordination. In addition, I wish to thank the HPC and Research Support Group in QUT for providing computational resources and services for this research. I would also like to thank Professional Editor, Dr. Adele Fletcher, for providing copy-editing and proofreading services according to the guidelines laid out in the University-endorsed national policy guidelines. My appreciation is further extended to the Australian Government and QUT for providing me financial assistance during my candidature.

I would like to express my warm and sincere gratitude to Madara Karunaratne, Anuruddha Rathnayake, Palmo Thinley, Dinesha Chathurani and Kasun Chinthaka for their constant support and valuable help. This support and help during the ups and downs of PhD candidature, was extremely important to me.

I would also like to thank my brother and sister for their support and laughter, given to me over the years. Their encouragement and support is always there to make me be a better person. Last but not least, I would like to express my gratitude to all my relatives, school and university teachers, friends and others who have helped me directly or indirectly along this journey.

ANATHTHA PATHIRANAGE DHANUSHKA KRISHNAJITH

Queensland University of Technology

October 2014

Chapter 1

Introduction

This thesis focuses on parallelization and memory management in Data Intensive All-to-all Comparison (DIAC) in limited memory conditions. In an *all-to-all comparison*, every item in a group of data items is pair-wise compared with the rest of the items in the group. An *all-to-all comparison* performed on a large number of data items is emphasised by the term "data intensive" in DIAC.

DIAC is a class of problems found in many fields [Moretti et al., 2010]. For example, it is a common pattern that appears throughout science and engineering in various applications [Manber, 1994, Moretti et al., 2010]. It is also widely used in bioinformatics [Altschul et al., 1990, Yu et al., 2010a], biometrics calculations and data mining. In bioinformatics usually DIAC is performed on genomic sequences using a comparison function such as sequence alignment. In biometrics calculations the data items can be finger-print data or facial images, while the comparison can be a custom function such as face recognition or finger print recognition. The output of the comparison is a single value indicating the similarity between the compared pair of items.

While simple to state, achieving the optimum speed of DIAC in limited memory is not

a trivial problem to solve. The problem remains non-trivial even without considering the parallelization. The complication of solving the problem can be briefly explained as follows. In limited memory, the memory may not be sufficient to hold all data items at the same time. Therefore, some data items need to be swapped in and out of memory multiple times to complete the DIAC. Each transfer of data item to memory (load) causes an increase in the total execution time. Completing a DIAC with the minimum number of loads yields the optimum speed for the DIAC in limited memory. Solving this problem is challenging due to the large number of combinatorial choices over the life-span of a DIAC. More challengingly, the data sets used in DIAC are huge. For example, genomic sequences in the composition vector method [Yu et al., 2010a] typically consist of hundreds of thousands input data items.

The overall aim of this thesis is to develop a systematic approach to significantly reduce the runtime of DIAC in limited memory conditions. To achieve this goal both efficient memory management and parallelization[1] in shared-memory computers are used. In the process, performance-related issues in DIAC in limited memory are identified and strategies are developed to overcome the identified gaps in memory management and parallelization. Then algorithms are designed and implemented to significantly improve the performance of DIAC over the existing methods. The parameter settings for the proposed algorithms are theoretically derived, and close-fitting performance-related theoretical lower and upper bounds for DIAC in limited memory are established. This research uses bioinformatics applications with DIAC such as the composition vector (CV) method [CVTree, 2011, Yu et al., 2010a] for case studies and examples.

The rest of this chapter is organized as follows. Section 1.1 discusses the research background followed by the motivation and significance of the research in Section 1.2. Section 1.3 presents statement of the research problem. Contributions of this work are

[1]Parallelization refers to the process of converting or completely rewriting a computer application designed to run sequentially, into a version which can exploit the power of a parallel computer.

claimed in Section 1.4. Section 1.5 and 1.6 outline the thesis, and scope and limitations of this research, respectively. Section 1.7 lists the publications during the PhD research.

1.1 Research Background

The result of an *all-to-all comparison* can be presented in a matrix format which is known as the correlation matrix or similarity matrix. Every value in a correlation matrix indicates the output of the comparison function for a pair of data items. Figure 1.1 shows a sample correlation matrix resulting from an *all-to-all comparison* performed on five data items. The correlation matrix considered in this research is a symmetric matrix (as shown in Figure 1.1) because the pair-wise comparison is assumed to be commutative.

	1	2	3	4	5
1	0.0	1.1	4.4	3.3	0.2
2	1.1	0.0	2.1	1.8	1.3
3	4.4	2.1	0.0	1.2	0.5
4	3.3	1.8	1.2	0.0	3.7
5	0.2	1.3	0.5	3.7	0.0

Figure 1.1: Correlation matrix based on *all-to-all comparison* of five data items. The value of each element represents the dissimilarity between a pair of data items.

The DIAC problems addressed in this thesis consist of a non-trivial preprocessing, which is conducted on the data items before the pair-wise comparison. The computational cost of a preprocessing of an item is significant relative to the cost of a pair-wise comparison. For instance, in the bioinformatics CV method [Yu et al., 2010a], the *composition vectors* are generated by preprocessing genomic sequences (data items)

and the pair-wise comparisons are carried out between the *composition vectors* (preprocessed data items). The method for calculating a *composition vector* based on a genomic sequence is a computationally intensive process.

After preprocessing, the preprocessed data items are stored in memory. Only the preprocessed data items in memory can be compared. In some applications, the preprocessing phase mainly involves reading the data items from the disk to memory. In this research, we abstract this preprocessing process as a significantly time-consuming operation to bring the data items to memory for the comparisons. We call this abstracted process as LOAD in algorithms and refer to it as *load* in text references.

The *load* process is not specific to each comparison. Therefore, while the preprocessed data items are stored in memory, they can be reused for the comparisons related to the data item. By maximizing these reuses, redundant preprocessing of data items can be minimized in limited memory conditions.

Each data item in DIAC applications addressed in this thesis requires a significant amount of memory to store after the *load* operation. The preprocessed data items can be around 100 MB to 1 GB each (e.g. in CV method [Yu et al., 2010a]) and there can be hundreds or even thousands of such data items for an all-to-all comparison problem. The total size of such data sets in memory often exceed the memory capacity of a typical shared-memory computer which typically ranges from 4 GB to 64 GB.

The tasks of DIAC (i.e. comparisons and *loads*) are independent from each other and can be executed in parallel. In fact, comparisons depend on *loads* although no comparison depends on each other and no *load* depends on each other. The tasks are assumed to be non-separable further into subsections which can be executed in parallel. However, a DIAC problem is still a good candidate for parallel execution because of the independent nature of the tasks.

If the time spent on bringing data items to memory (*load*) in an *all-to-all comparison* is negligible, the problem becomes embarrassingly parallel. However, if memory capacity limits the number of data items stored in memory, the problem may not be embarrassingly parallel. This thesis focuses on the problems where a *load* operation is significantly time consuming relative to a comparison, not when time spent on bringing data items to memory is negligible.

Minimizing the total execution time of DIAC in shared-memory systems using parallelization is challenging because of the *load* operations and the limited memory. Redundant *loads* caused by the limited memory can increase the runtime due to the time spent on *loads*. On the other hand, the limited number of data items in memory also limits the number of available comparisons which can be executed in parallel. In addition, the number of comparisons available in memory depends on the combination of data items in memory as the data items loaded to memory can be shared between processors in shared-memory computers. The time required for *loads* limits the rate of incoming comparisons which eventually limits the rate of completing comparisons.

In summary, parallelization of DIAC with time-consuming *load* operations in memory constraints is challenging. A combinatorial number of ways can be found to complete all comparisons in a DIAC over the life-span of the application. A good solution in this situation should not only manage memory efficiently but also look into increasing the number of parallel tasks throughout the calculation. Therefore, finding the best schedule for executing *loads* and comparisons to minimize the total execution time is a hard problem to solve. This is the main theme of this thesis.

1.2 Motivations and Significance

From the bioinformatics perspective, our research has a potential to make a significant contribution towards reducing the runtime of many bioinformatics applications. DIAC can be found in many bioinformatics applications such as the CV method [Yu et al., 2010a] and BLAST [Altschul et al., 1990]. Many of these methods share the same pattern of *all-to-all comparison* with a preprocessing stage before the comparisons, including the CV method and its variations listed by Wang [2009]. All proposed solutions in this thesis are based on this basic pattern. Therefore, the solutions are immediately beneficial to many DIAC applications based on similar pattern, such as CVTree [CVTree, 2011].

Our motivation to this work is increasing the speed (decrease runtime) of two recent applications by using parallelization. The application developed by Yu et al. [2010a] comprises of a DIAC phase as the first step towards generating a phylogeny tree [Feng and Doolittle, 1990] for a given set of genomic sequences. Since their main interest is in the final results, the memory and computing resource management of the application has not received much attention. For this reason, the application performs many redundant *loads* even when a large memory is present, eventually causing a significantly long runtime. Another application called CVTree [CVTree, 2011] uses an improved memory management algorithm. It does not have the support for parallel execution.

Furthermore, in the applications developed by Yu et al. [2010a] and CVTree [Xu and Hao, 2009], higher-order calculations are prohibitively time-consuming and sometimes impossible due to their much higher demand for resources. Yu et al. [2010a] suggest that higher-order pair-wise comparisons may give better results than lower-order calculations. By using our new algorithms it is possible to run higher-order calculations significantly faster than their original applications.

Parallelization of memory-constrained DIAC has only received a little attention in existing literature. The limited existing research reports on the problem of limited memory such as Wu et al. [2009] only present theoretical complexity analyses of the problem. However, due to the added complexity of managing limited memory, the existing parallelization techniques are rendered ineffective. In addition, the unpredictability of *load* and comparison times does not favour many existing scheduling algorithms as most of them rely on accurate estimating of task runtimes. To the best of our knowledge, research reports do not exist in the literature to propose solutions specifically for parallelization of memory-constrained DIAC in shared-memory multi-core platforms.

Tools to visualize and simulate the behaviour of DIAC are important for analysing purposes. Without such tools, it is hard to analyse and identify the problems in managing memory for DIAC. This is for two reasons,

- While developing parallel algorithms, it is important to verify the intended behaviour by using visualizations.
- Since most of the applications used as examples in this research have prolonged execution times, simulations are essential for developing the algorithms as well as for the evaluation purposes.

Simulations are also used for the parameter predictions in the proposed parallel algorithms. Therefore, the simulations are used as a part of scheduling algorithm and must be lightweight and extremely fast not to hinder the performance of the scheduling algorithms. To validate the theoretical results of this thesis, simulators should be able to simulate theoretical behaviour of the proposed algorithm. Therefore, developing such tools is beneficial not only to our research, but also to other research studies conducted in the similar context.

1.3 Statement of the Research Problem

The main objective of this research is: *minimizing the runtime of memory-constrained, data intensive all-to-all comparison for uni-processor and shared-memory multi-core/multi-processor platforms by efficiently utilizing available computing resources.*

The research questions addressed in this thesis are:

Research Question 1: How should memory be managed for DIAC in single-core (uni-processor) platforms to minimize the time spent on I/O (*load*) operations?

Research Question 2: How can DIAC be parallelized in limited memory in conjunction with memory management, while keeping a proper balance between the overhead of I/O operations and speed gain from parallel execution?

Research Question 3: How may the proposed algorithms be evaluated, their behaviour analysed under various conditions (dataset and system properties) and theoretical results validated?

1.4 Main Contributions

To answer the three questions presented in Section 1.3, this thesis makes three major contributions:

Contribution 1: A novel memory management (paging) algorithm is proposed to significantly minimize the time spent on *loads* of data items to memory. It is empirically demonstrated to substantially improve the computing performance over the existing approaches (from 7% to 31.9 times improvement in speed). Further, it is demonstrated to use minimum (optimum) number of *loads* by using a brute-force technique for small number of data items to be compared. The

optimum parameters for the algorithm are derived theoretically. A close-fitting lower bound for the minimum number of *loads* required to complete a DIAC is also theoretically derived.

Contribution 2: By incorporating dynamic scheduling with static scheduling, a novel parallelization technique is introduced. It utilizes a problem-specific pattern developed for the targeted abstracted problem. This technique significantly minimizes the scheduling overhead of complex parallelization problems where significant prior knowledge of the abstracted problem is available (e.g. data dependencies and tasks), while still producing dynamic schedules with good quality. Implementing this technique, a novel distributed parallelization algorithm is proposed to significantly reduce the runtime of DIAC in limited memory through maximum utilization of available memory resources and computing power. The algorithm targets shared-memory multi-core computers. It dramatically speeds up existing bioinformatics applications (e.g. 7.86 speed-up in an 8-core shared-memory system and 15.77 speed-up in a 16-core shared-memory system) compared to their sequential counterparts while efficiently managing the memory within the specified limitations. It is empirically shown to attain close to the maximum utilizations of processors for comparisons. From our simulation studies, the optimum parameters for the algorithm are theoretically derived for typical scenarios commonly found in many applications.

Contribution 3: To assist evaluating the performance of parallel algorithms, including the one proposed in this thesis, a novel simulator and a parallel framework for DIAC are developed. The simulation environment is designed and implemented from scratch for research on shared-memory multi-processor systems, in order to exploit the advantage of the assumed uninterrupted and continuous processor-task association of the proposed parallelization algorithm. It can be used to simulate other parallelization algorithms with similar properties as well. The simulation tool is light-weight and extremely fast to simulate long running applications. It is used to simulate the developed algorithms, theoretical re-

sult validations, optimum parameter prediction in runtime and to observe the algorithms' behaviours in various conditions. The novel parallelization framework assists rapid implementation of parallel algorithms developed for DIAC and clearly separates the decision-making logic from other implementation details.

1.5 Thesis Outline

The rest of this thesis is organized as follows.

- Chapter 2 is a comprehensive review of the existing research in related research fields. It identifies and justifies the research context from which the research questions have been derived.
- Answering the first research question (Contribution 1), Chapter 3 develops an algorithm to manage memory for *all-to-all comparison* in single-core (uniprocessor) systems. A close-fitting lower bound for the minimum number of required *loads* to complete a DIAC and optimum parameter settings for the proposed algorithm are theoretically derived. It also compares existing memory management strategies with the proposed memory management algorithm.
- Chapter 4 is a preparation to solve our DIAC problem in Chapter 5 and 6. In this chapter, the DIAC parallelization problem under memory limitations is modelled. An upper-bound for the maximum parallel speed-up of DIAC under limited memory is also derived. In addition, the proposed model is extended to adapt existing parallelization techniques to solve our DIAC problem.
- Answering the second research question (Contribution 2), an algorithm for parallelization of DIAC with memory management in multi-core processors is developed progressively, utilizing a novel parallelization technique in Chapter 5. Optimum parameter prediction techniques for the proposed algorithm are also

theoretically developed.

- The algorithm developed in Chapter 5 is extended in Chapter 6 to address the practical and implementation issues. The experiments for performance validation and behaviour analysis of the proposed algorithm are also conducted in this chapter.
- Answering the third research question (Contribution 3), the simulation environment which is designed and implemented to assist our research and the parallel algorithm is discussed in Chapter 7. It also describes the novel parallelization framework which assists rapid implementation of parallel algorithms for DIAC. A unique visualization tool developed to assists our research for analysing the runtime behaviour of DIAC (using new graphical representations for DIAC) is also discussed in this chapter.
- Finally, Chapter 8 concludes the thesis.

1.6 Scope and Limitations of the Thesis

This research targets shared-memory multi-core/multi-processor platforms for parallelization of DIAC. The novel pattern-based parallelization technique which will be proposed in the thesis has the potential to be used in other platforms as well, although this capability is not demonstrated in this thesis. The DIAC-related memory management and parallelization problems found in other computing platforms such as distributed computing systems are out of the scope of this thesis.

In the DIAC problems considered in the thesis, the time spent on the *load* operation (includes preprocessing) has to be significant compared to the time spent on comparisons. Although the algorithms are aimed to speed-up DIAC applications with such significantly time-consuming *load* operations, they can still be used to speed-up other DIAC applications as well.

This research does not assume the size of data items as a constant. Although some intermediate algorithms assume that the size of data items is constant, an extended version to handle variable sized data items is always presented. Similarly, the *load* and comparison times are not assumed as constants. However, the theoretical results assumes that the sizes of data items, size of memory and task runtimes are as constants. The theoretical results are still useful with average values of variable data item sizes and task runtimes.

1.7 Publications During PhD Research

During the PhD research, three papers have been published, including two journal papers and 1 conference paper. They are listed as follows.

1. Krishnajith, A. P. D., Kelly,W., and Tian, Y.-C. (2014). Optimizing I/O cost and managing memory for composition vector method based correlation matrix calculation in bioinformatics. *Current Bioinformatics*. In press. — From the findings in Chapter 3.
2. Krishnajith, A., Kelly, W., Hayward, R., and Tian, Y.-C. (2013). Managing memory and reducing I/O cost for correlation matrix calculation in bioinformatics. *In proceedings of the 2013 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, pages 3643. — From the findings in Chapter 3.
3. Han, G. S., Yu, Z. G., Anh, V., Krishnajith, A. P. D., and Tian, Y.-C. (2013). An ensemble method for predicting subnuclear localizations from primary protein structures. *PLoS ONE*, 8(2):e57225. — Based on some of the findings in Chapter 3.

Chapter 2

Literature Review

This chapter reviews the literature related to *Data Intensive All-to-all Comparison (DIAC)*, memory management of DIAC, parallelization of DIAC and existing simulation techniques. We start by investigating the properties of DIAC addressed in this research and existing applications of DIAC. Then we discuss the existing memory management techniques and their findings for solving DIAC in limited memory. Thereafter, existing solutions for the DIAC parallelization problem are examined. The popular parallel computing strategies which were not proposed specifically for parallelization of memory-constrained DIAC but have the potential of solving the problem are discussed next. Then the various parallel computing platforms are briefly discussed with their properties and differences, to clearly distinguish the platform-specific variations of strategies in parallelization. Thereafter, the existing simulation tools are reviewed to identify a suitable simulation platform for our research. Finally, the literature reviewed in the chapter is summarised.

2.1 All-to-All Comparison

This section extends the discussion in Chapter 1 on the properties and applications of *Data Intensive All-to-all Comparison (DIAC).*

Algorithm 1 and 2 are two algorithms to perform a DIAC. In the algorithms, DIAC is performed on a group of N data items. G_i denotes each data item in the group and V_i denotes the loaded (i.e. preprocessed) G_i in memory, where $0 \leq i < N$. The correlation matrix is denoted by $C[i, j]$, where $0 \leq i, j < N$.

Algorithm 1 A procedure to complete a DIAC by loading all data items into memory and performing a minimum number of loads

```
1:  procedure DIAC1
2:      for i = 0 → N − 1 do
3:          V_i = LOAD (G_i)
4:      end for
5:      for i = 0 → N − 1 do
6:          for j = i + 1 → N − 1 do
7:              C[i, j] =COMPARE(V_i, V_j)
8:              C[j, i] = C[i, j]
9:          end for
10:     end for
11: end procedure
```

In Algorithm 1, all data items are first loaded into memory and then the pair-wise comparisons are carried out between the items in memory. This algorithm requires sufficient memory to *load* and keep all data items in memory. However, in the DIAC problem addressed in this thesis, memory is insufficient to hold all data items. Therefore, Algorithm 2 only keeps maximum of two data items in memory at a time. This algorithm:

1. *loads* one data item;
2. then loads the rest of the data items (not compared earlier) one by one, compares the new data item to the one in the memory, and then unloads the new data item;

3. unloads all data items and
4. repeats the steps for every data item.

Even though this algorithm's memory requirement is equal to the size of two data items, it performs a large number of *loads*.

If the *loads* are carried out for each comparison (not sharing loaded data items) as in Algorithm 2, the complexity of the loading phase is $\mathcal{O}(N^2)$, where N is the number of data items to be compared. Since the *loads* are significantly time consuming, the time spent on this phase can be saved if every loaded item (V_i) can be kept in memory for later use as in Algorithm 1. The complexity of the loading phase in Algorithm 1 is $\mathcal{O}(N)$.

Algorithm 2 A procedure to complete a DIAC by loading a data item to memory for every comparison and using a minimum amount of memory

```
1:  procedure DIAC2
2:      for i = 0 → N − 1 do
3:          V_i = LOAD (G_i)
4:          for j = i + 1 → N − 1 do
5:              V_j = LOAD (G_j)
6:              C[i, j] =COMPARE(V_i, V_j)
7:              C[j, i] = C[i, j]
8:              UNLOAD (V_j)  // Delete V_j from memory
9:          end for
10:         UNLOAD (V_i)
11:     end for
12: end procedure
```

When the data items cannot all be held in memory at the same time, we have two choices:

1. We can write the preprocessed data items (V_i) out to disk before the comparisons begin and then read them back in prior to their use.
2. Alternatively we can re-compute composition vectors each time they are required.

Which of these approaches we should use depends on the cost of re-computing a composition vector compared to the cost of reading it from disk. In most cases, reading pre-calculated composition vectors from disk will be more efficient, but to avoid confusion, we use "LOAD" in our algorithms and "*load*' in text, to indicate that a composition vector is brought into the memory either by reading the disk or generating it from the scratch. It is important to note that the algorithms proposed in this thesis do not depend on this choice.

Typically, data items are required to be preprocessed at least once (where it can be written out to disk), as a requirement of memory management. The reason is that the memory requirement of preprocessed data items cannot be predicted in calculations such as the composition vector method. More details on utilization of the disk will be discussed in Section 3.4 and 5.2.

2.1.1 All-to-All Comparison in Composition Vector Method

This thesis uses Composition Vector (CV) method [Yu et al., 2010b] in bioinformatics for case studies and examples. Therefore, this section concentrate on the details of specific properties and attributes of DIAC in the CV method. However, it is noted that the solutions proposed in this thesis are applicable to many DIAC problems with similar properties. Some of such problems will be discussed in Section 2.1.2.

We introduced the correlation matrix briefly in Chapter 1. The correlation matrix has following properties in composition vector (CV) method in bioinformatics. If the correlation matrix is $C[X, Y]$ which represents the correlation between the genomic sequences X and Y, the following set of properties can be identified in it.

1. $C[X, Y] = C[Y, X]$
2. $C[X, Y] \geq 0$; and $C[X, Y] = 0$ if and only if $X = Y$

The correlation matrix $C[X, Y]$ is said to be normalized if $0 \leq C[X, Y] \leq 1$ [Yu et al., 2010b]. Let $\{G\} = \{G_i : i \in \mathbb{Z}, 0 \leq i < N\}$ be a set of genomic sequences. The pairs, (G_i, G_j) are compared to generate the distance matrix for the genomic sequence set, for $0 \leq i, j < N$. According to above rule 1, $C[i, j]$ is a symmetric matrix with the distance (correlation) of (G_i, G_j) is at $C[i, j]$ and $C[j, i]$. According to rule 2, the diagonal of the matrix will always be zero. This leaves only a section of the matrix $C[i, j] = correlation(G_i, G_j)$ where $0 \leq i, j < N$ and $i < j$, to be calculated. This is a triangular region of pair-wise distances, similar to the correlation matrix shown in Figure 1.1.

In the correlation matrix C, $C[i, j]$ is an indication of the similarity between genomic sequences G_i and G_j. To find this similarity, there are two main categories of comparison methods, alignment-based and alignment-free methods [Wang, 2009]. In the alignment-based methods, to compare two sequences of length l, the computational cost and the memory requirement are both $\mathcal{O}(l^2)$ [Waterman, 1995].

Due to the higher demand of computation and memory resources to align these gnomic sequences using alignment-based methods, alignment-free methods are developed for the whole genome phylogeny instead of alignment-based methods. Alignment-free methods can be divided into three classes; which are gene content method, data compression method and Composition Vector (CV) method [Wang, 2009].

Phylogenetic analysis is an important problem in bioinformatics which involves an DIAC. The phylogenetic tree, which is a hierarchical tree, can be calculated based on the pair-wise phylogenetic distances in the correlation matrix (distance matrix) [Rajasekaran et al., 2005]. It is used to analyse the phylogenies in various species and phylogenetics is the study of evolutionary relationships among various groups of organisms. Phylogenies are discovered through molecular sequencing data and morphological data of the species [Wang, 2009].

The DIAC for large sized genomic sequences is a part of the CV method. In this method, the *composition vectors* are constructed for each species based on their whole genomic sequences. Then, the distance between the *composition vectors* is used as the distance between species.

The CV method has achieved a great success recently according to Wang [2009]. It was proposed by Hao et al. [2003] for the whole-gnome-based prokaryotic phylogeny. Their phylogenetic tree, generated by this method, provided a classification of the three domains of life and the classification is consistent with those based on traditional analysis. Since this method was successful, quite a few models have been proposed in this direction [Chan et al., 2010]. Basically all models that use the CV method, have the following steps in common.

1. Construction of the *frequency vectors* — This vector consists of the number of occurrences of patterns in a sequence. The *frequency vector* is calculated as follows. Consider a genomic sequence of length l. A window with length K where $1 \leq K \leq l$ is slid through the sequence and frequency of sub-strings of size K (K*-strings*) is recorded in the *frequency vector*. The length of the composition vector depends on the number of nucleotides in the sequence. The number of nucleotides can be either 4 or 20. As a result, the length of the frequency vector is either 4^K or 20^K [Wang, 2009].
2. Construction of the *composition vectors* — The *composition vector* is calculated based on the *frequency vector*. This is a vector which represents the corresponding species. The evolution distance between the species can be measured directly by the distance between their *composition vectors* [Wang, 2009]. Therefore, in the CV method the comparison between genomic sequences are conducted between the *composition vectors*.
3. *All-to-all comparison* of *composition vectors* — The correlation matrix calculation is conducted in this phase by carrying out the pair-wise *all-to-all* compar-

isons.

4. Construction of the phylogenetic trees — The phylogenetic tree, based on the correlations calculated in step 3, is built in this phase.

Various ways exist to estimate the noise in step 2 and to take distance measures in step 3. However, the basic steps in each method remains the same as listed by Chan et al. [2010]. An application developed to calculate the correlation matrix should have the three basic steps as listed below which correspond to the above mentioned steps in CV method (e.g. Application written by Yu et al. [2010a]). These steps are briefly described below.

- Step 1—The data from the genomic sequences are loaded into memory (Random Access Memory). This step includes Disk I/O (Input/Output).
- Step 2—The *composition vectors* are calculated and stored in memory. Storing these long vectors require a significant amount of calculations and sufficient memory to store them.
- Step 3—*All-to-all comparison* is performed on the *composition vectors* to build the correlation matrix.

Step 1 and 2 can be considered as a single phase where genomic sequences are *loaded* into the memory. Since Step 2 is a computationally intensive procedure, it is usually faster to write the *composition vector* to the disk and load it whenever required.

2.1.2 Other All-to-All Comparison Problems

Genome assembly [Havlak et al., 2004, Huang et al., 2003] remains one of the most challenging computational problems in bioinformatics [Moretti et al., 2010]. The output of a sequencing device is used to produce genomic sequences. However, these devices can only produce overlapping sub-strings of the targeted genomic sequence

due to physical constraints of the device. The assembler must align all these hundreds of thousands of sub-strings and produce the targeted sequence. The process involves an *all-to-all comparison* of the sub-strings followed by several data grouping stages [Corpet, 1988, Gotoh, 1993, Miller, 1993, Moretti et al., 2010].

Multiple Sequence Alignments (MSA) [Barton, 1990, Feng and Doolittle, 1987, Higgins and Sharp, 1988, Jaap and Heringa, 1999, 2002, Notredame et al., 2000, Schuler et al., 1991, Thompson et al., 1994] are an essential method for protein structure and function prediction, and other common tasks in sequence analysis [Edgar and Batzoglou, 2006]. The first stage for solving an MSA includes a calculation of the correlation matrix between each pair of sequences. It is followed by determining the alignment topology, and finally solving the alignment of sequences or clusters themselves [Rizk, 2005]. This correlation matrix calculation involves a DIAC. These methods yield useful results, though they are computationally intensive [Date et al., 1993].

Moretti et al. [2010] have shown the *all-to-all comparison* problem as a frequently found problem in biometrics, data mining and bioinformatics [Moretti et al., 2010]. These applications were briefly discussed in Chapter 1.

According to Moretti et al. [2010], the Face Recognition Grand Challenge [Phillips et al., 2005] is a typical DIAC problem in biometrics. The problem is to compare 4,010 images, each from the Face Recognition Grand Challenge [Phillips et al., 2005] to all others in the set, using functions that range from 1 to 20 seconds of compute time, depending on the algorithm in use [Moretti et al., 2010].

2.1.3 All-to-All Comparison in Bioinformatics Applications

We currently have access to source codes of two recently written applications of DIAC using the CV method Yu et al. [2010a]. This section reviews these two applications.

The performance evaluations of our proposed methods are based on these two applications. The two applications take a set of large (long) genomic sequences (DNA/RNA or Peptide) as the input to DIAC. It is important to note that the two applications discussed in this section are developed primarily to demonstrate the bioinformatics methods proposed by the authors of the applications. Optimizing these applications using parallelization or efficient memory management has not been their primary concerns in the implementations.

The application developed by Yu et al. (2010b) is written to load two genomic sequences into the memory at a time, compare them, and then go to the next pair. The process is repeated for every comparison. Thus, only two genomic sequences are kept in the memory at a time, bearing in mind that a *composition vector* is created for each genomic sequence before the comparison is completed. In this application, the length of the composition vector is 4^K (for DNA/RNA sequences) or 20^K (for peptide sequences) where K is the order of the calculation [Yu et al., 2010b]. When the order is 6 ($K = 6$), the memory required for a composition vector of a single genomic sequence is approximately 250Mb (4×20^6 bytes). This application by Yu et al. [2010a] is unable to fully utilize the available memory to make the comparison faster by keeping more genomic sequences in the memory. In addition, it does not support for parallel execution and cannot take the full advantage of multi-core processors.

The second application is from the CVTree website [CVTree, 2011, Xu and Hao, 2009] and has two versions. In one version, all genomic sequences are loaded into the memory and the comparisons are completed. Since this program may exceed available memory capacity when the number of sequences is large, they developed another version written to support a DIAC of a large number of genomic sequences in limited memory. The later version is considered in this review since we are interested only in situation where memory is insufficient to hold all genomic sequences.

In the later version of the CVTree [2011] applications, there are a few important de-

sign decisions made by the authors. In the application, *composition vectors* are written into the disk and retrieved later from the disk when needed. The authors [Xu and Hao, 2009] have used a more advanced memory management algorithm than Yu et al. [2010a]. The application takes a memory limit from the user as an input and is supposed to run within the specified memory limit. It holds more than two genomic sequences in memory at a time, and takes the advantage of the available physical memory to make the comparison faster by reducing the loads. However, because of the problems of the memory management algorithm, the CVTree [2011] application tends to use memory over the user-specified limit. Furthermore, this program does not support parallel execution. It is designed to be executed on personal computers, but it is unable to fully utilize the processing power of recent multi-core computers due to lack of parallelism.

2.2 Memory Management in Single-core Computers

Wu et al. [2009] analysed a similar problem for the memory-constrained DIAC in single-core processors. They proved that their problem is NP-complete. Although their problem occurs in grid computing systems, it shares some properties of our DIAC with limited memory. In their system, a group of jobs are dispatched to a uni-processor node. The jobs share remotely stored data. To complete a job, the required data must be received by and available in the working node before the job can commence. The working node has a limited capacity to store received data, and the previously received data may be deleted to accommodate new data.

Similarities between our DIAC problems with the problem addressed by Wu et al. [2009] are shown in Table 2.1 and the differences are shown in Table 2.2. Even though the two problems share many similarities as seen in Table 2.1, the differences seen in Table 2.2 distinguishes the two problems. Therefore, the NP-completeness proof by

Wu et al. [2009] for their problem where every job depends exactly on two data items, does not prove that our problem is NP-complete.

Wu et al. [2009] prove that finding a Hamiltonian edge sequence in a graph is NP-complete. This proof can be extended to our problem to prove that our problem is NP-complete, if each comparison and *load* requires one time unit to complete and the number of data items memory can hold is two (i.e. $M = 2$). However, under these condition our DIAC problem is unrealistic. Therefore, the NP-completeness of our DIAC problem still remains open for investigation.

Table 2.1: Similarities between the problem addressed by Wu et al. [2009] versus our DIAC problem

Wu et al. [2009]	**DIAC**
Relevant data must be available in the node to start a job.	Relevant data must be available in the memory to start a job (comparison).
Data is received by the node from a remote server. This process is significantly time consuming.	Data is loaded to the memory by reading the disk or by generating from the scratch (the *load* process). Both processes are significantly time consuming.
There is an upper bound for the capacity to store data in the node.	There is an upper bound for the capacity to store data in the memory.
Files loaded to the node are shared between jobs.	Data items loaded to the memory is shared between jobs (comparisons).
The objective is to schedule all jobs within a minimum timespan.	The objective is to schedule all jobs (comparisons) within a minimum timespan.

Table 2.2: Differences between the problem addressed by Wu et al. [2009] versus our DIAC problem

Wu et al. [2009]	**DIAC**
Jobs with all data item available in the node can be completed while data is being received	Both comparisons (jobs) and *loads* has to be carried out sequentially one after another since data items needs to be preprocessed before comparisons
Time required to complete a job and to receive a data item are same (i.e. 1 unit of time)	Time required to complete a comparison and to complete a *load* can be unequal.

Without considering the limited capacity scenario, Giersch et al. [2004] investigate a similar problem to Wu et al. [2009]. Interestingly, they prove that, even if memory is considered to be unlimited, the problem is still NP-complete when the jobs are independent from each other and depend on at least three data items. Wu et al. [2009] prove that there exists an optimum algorithm if the memory is unlimited and every data item depends on only two data items. They present an algorithm for this scenario.

Mueen et al. [2010] have proposed a heuristic, called "Optimal Baseline Caching Algorithm", to solve the scheduling of DIAC in limited memory. In their algorithm, all items to be compared are assumed to be uniform in size and the maximum number of items fit in the memory is assumed to be a constant (i. e. M and N are constant). This algorithm is depicted in Algorithm 3.

Algorithm 3 Memory management algorithm proposed by Mueen et al. [2010].

```
1:  procedure MUEENMETHOD
2:      for i = 0 → N − 1 step M − 1 do
3:          A ← {G_x : x ∈ ℤ, i ≤ x < Min(i + M − 1, N)}
4:          LOAD (G_A)
5:          for j = i + M − 1 → N − 1 do
6:              LOAD (G_j)
7:              COMPAREALL (A ∪ G_j)
8:              UNLOAD (G_j)
9:          end for
10:         UNLOAD (G_A)
11:     end for
12: end procedure
```

Our memory management problem can also be modelled as a traditional virtual memory management problem [Aho et al., 1971]. The data items can be considered as pages and memory can be divided into frames to hold them. The memory frames are insufficient to hold all pages. The objective is to intelligently replace the unused pages to reduce the number of misses in the future. There are many frequently used generic page replacement algorithms such as Least Recently Used (LRU) [Aven et al., 1976], Least Frequently Used (LFU) [Li et al., 2008], First In First Out (FIFO). There has

been a widely accepted "informal principle of optimality" for the page replacement policies: the page to be replaced is that which has the longest expected time until next reference [Aho et al., 1971]. Accordingly, if an algorithm knows exactly which page is going to be referenced last, it can handle page replacement optimally. A memory management algorithm which uses the prior knowledge of the memory access patterns of an application is aware of the best page to be replaced next. Without this knowledge, any generic paging strategy (such as LRU) cannot hope to compete against an algorithm such as that developed by Mueen et al. (2010).

Unlike the above mentioned direct memory management strategies, job pruning is an indirect strategy to reduce the burden of *load* operations [Agrawal et al., 1993, Mueen et al., 2010, Zhu et al., 2002]. In this approach predetermined unnecessary comparisons are pruned, thereby reducing the cost of *load* operations. This is achieved by predicting uncorrelated pairs, utilizing a certain threshold and using special pre-calculations before calculating the accurate correlation. While this kind of prediction works well for correlation-based similarity searches, it is not always applicable.

2.3 Specific Memory Improvements for the CV Method

The memory complexity of storing a *composition vector* without any optimizations is $\mathcal{O}(n^K)$ where n is the size of the alphabet of the genomic sequence and K is the order of calculation in the CV method. Typical values for n are 4 or 20 depending on the type of the genomic sequence (DNA or protein sequence) and K ranges typically from 6 to 20 [Yu et al., 2010a]. Due to significantly large memory usage, specific memory optimization techniques have been proposed to make the CV method efficient. Some of these techniques are used in our benchmark programs [CVTree, 2011, Yu

et al., 2010a] together with our memory management algorithms for much enhanced computing efficiency.

Wang [2009] proposed a method in which the memory required to store a *composition vector* is $\mathcal{O}(l)$, where l is the length of the genomic sequence. He used a sparse data structure called tables to store the index and frequency in columns. Since a typical *composition vector* mostly contains zeros, this method reduces the size of the *composition vectors* by sparing the memory allocated for zeros. In addition to the reduction of the memory required to complete a pair-wise comparison, this sparse structure increases the speed of a comparison.

Steinbiss and Kurtz [2012] proposed another memory optimization, which is independent of the programming language and can be used generally to store genomic sequences. Their space-efficient data structure, GtEncseq, can be used to store multiple biological sequences of a variable alphabet size. It includes customizable character transformations, wild-card support, and an assortment of internal representations optimized for different distributions of wild-cards and sequence lengths [20].

While the above strategies help minimize the amount of data required to store data in memory, the problem of being unable to *load* all the *composition vectors* into the memory is not completely solved, particularly when N, the number of genomic sequences is large. This thesis will address this problem.

2.4 Parallelization of All-to-All Comparison

There are many solutions proposed for the parallelization of DIAC in various computing platforms. DIAC falls into the application category called *data intensive bag of tasks* [Lee and Zomaya, 2006]. The applications in this category consist of a set

of independent tasks which share data among them. Only a few solutions have been proposed for the parallelization of *data intensive bag of tasks*. Those solutions are also valid for the parallelization of the DIAC problem as well. Therefore, the solutions proposed directly for parallelization of DIAC are discussed in the following subsection (2.4.1). InSection 2.4.2, solutions proposed for the parallelization of *bag-of-tasks* and tasks sharing data are discussed. The research discussed in this section are not limited to a shared-memory platform.

2.4.1 Parallelization Specific to All-to-All Comparison

The earliest research to parallelize DIAC was carried out by Date et al. [1993]. Focusing on distributed memory systems, they proposed a farm-computing approach to execute the DIAC in parallel. In this approach, one processor (the farmer) break the large number of work-units into sub-sets and distribute the subsets to one of the other processor (the worker). A work-unit is a pair-wise alignment (comparison). In a set of N genomic sequences, $N(N-1)/2$ work-units need to be completed. The work is distributed dynamically. A work-unit is sent to a worker processor by the farmer processor, and the worker processor accepts the work unit, if it is free. The result of the alignment is sent back to the farmer processor upon completion. The worker processors do not communicate with each other and all the communications are routed through the farmer processor. Date et al. [1993] indicate two major reasons to use the farm approach for parallelization of this particular problem:

1. The time to process a single work unit is significantly larger than the time to generate and distribute work.
2. The order in which the results are collected is not important.

By using this approach, Date et al. have archived almost linear speed gain, when the number of transputers increased from 4 to 64. They do not consider any limitation in

memory in the transputers. The input data are not expected to cause significantly large communication cost.

In 2002, Kleinjung et al. [2002] proposed a solution for parallelization of Multiple Sequence Alignment (MSA). MSA has a DIAC as a part of the process. In their approach they use the Single Instruction Multiple Data (SIMD) architecture (refer to Section 2.6). Their approach is to distribute all genomic sequences across the networked computer nodes and let the nodes choose the tasks to execute. The pairwise comparisons are executed simultaneously in every node and managed using internode communications. They use the MPICH package [Lusk et al., 1996, Pacheco, 1997] to handle the parallel execution of the tasks. This work does not consider memory limitations in the nodes.

Another attempt of parallelization of DIAC of genome sequences targeted a cluster of computers [Hill et al., 2008]. A cluster of computers is a tightly coupled network of computers with distributed memories. To create batches of jobs to be distributed to the nodes, the correlation matrix is divided into rows. Since the correlation matrix is symmetric, the effective comparisons forms a upper triangular matrix. Therefore, the length of a row decreases gradually from top to the bottom of the matrix. Each row of comparisons is assigned to a node of the cluster. Since the batches of tasks (rows) are non-uniform, they are dynamically assigned to the nodes. Once the comparisons in a row are finished by a node, the next row is dispatched to the node. The approach has not addressed the scenario where the number of rows are less than the number of nodes in the cluster. Furthermore, it does not consider the opportunity to breaking the rows or tile the matrix. In this situation, some nodes will be idle since there are no rows to be assigned to the nodes. Furthermore, the approach does not take the memory limitations into account.

Moretti et al. [2010] proposed new strategies for parallelization of DIAC in campus grid environments. Targeting DIAC as an abstracted class of frequently found prob-

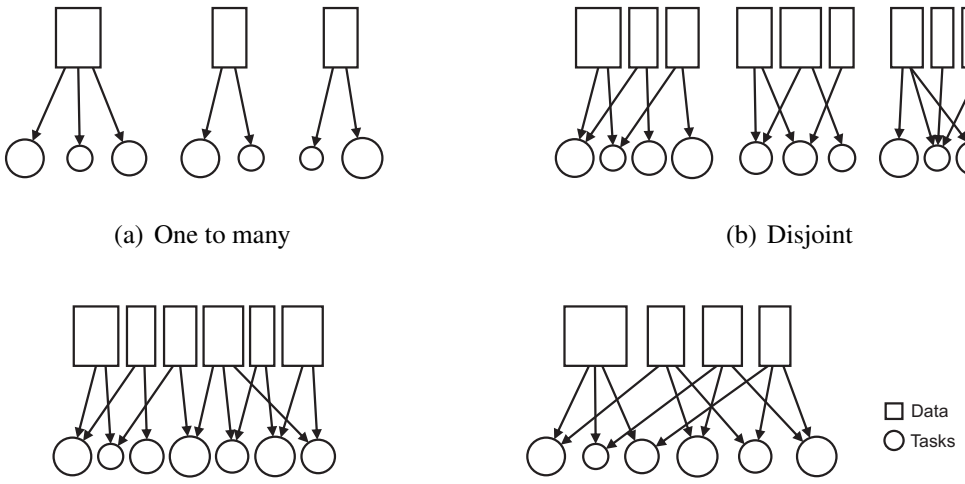
lems in many fields, they focused on both data distribution through the campus network and on the task allocation strategies. Since campus grids are extremely volatile and unpredictable, they addressed the reliability aspect as well. The conventional cluster schedulers usually distribute tasks in the same order as the tasks in the supplied or generated task queue. Instead, Moretti et al. proposed a new task organizer for both task and data distribution called the *All-pairs Engine*. The *All-pairs Engine* re-organizes the task queue and data distribution plan based on the data sharing patterns in the abstracted problem. The task and data distribution are also aware of the topology of the computer grid. Therefore, a new method called *topology-aware spanning tree* is also proposed in Moretti et al. [2010] for faster data distribution. This method significantly improves data distribution rate according to their reported experiments.

A recent attempt for parallelization of DIAC in a shared-memory system was made by Katoh and Toh [2010]. It parallelizes the DIAC as a part of the parallelization of an application for multiple sequence alignment (MSA), called *MAFFT* [Katoh et al., 2002]. In their particular attempt, the parallelization of DIAC phase is fairly straightforward mainly because the genomic sequences that they consider are small and the system's memory is sufficient to hold all of them at the same time. They have assigned each pair-wise comparison to a thread to calculate them simultaneously and independently.

Some of the attempts to parallelize DIAC, have investigated specific algorithmic parallelization techniques such as the work by Rajasekaran et al. [2005] and Pekurovsky et al. [2004], which are not commonly applicable in all of the problems. These solutions basically look for parallelism available within the pair-wise comparison algorithm which may not be generally applicable to all DIAC problems.

2.4.2 Parallelization of Tasks Sharing Data

When the input data is large, tasks sharing data is a *data intensive bag of tasks* problem. *Bag-of-tasks* is a set of independent tasks which share the input data. Lee and Zomaya [2006] have analysed the categories of *bag-of-tasks* problems depicted in Figure 2.1. In Figure 2.1, we have added our DIAC problem into the figure at the end of the categories listed by Lee and Zomaya. As seen in the figure, each task depends only on two data items in DIAC. Lee and Zomaya have shown that the tasks in the categories which they have listed (i.e. Figure 2.1:(a), (b) and (c)) can be often grouped based on the high affinity to certain input data. However, it is difficult to define such groups in DIAC, since every data item is connected to each other through a task.



(a) One to many (b) Disjoint

(c) Random (d) All-to-all comparison

Figure 2.1: Categories of *bag-of-tasks* problems [Lee and Zomaya, 2006]: (a) One to many; (b) Disjoint; (c) Random; and (d) All-to-all comparison.

Lee and Zomaya [2006] have also proposed a method for parallel execution of tasks sharing input data on grid systems. They specifically target the *data intensive bag of tasks* problem. The targeted platform is a grid system comprising of sites, each of which is a set of computational hosts participating in the grid. The method assumes that the hosts in the same site are able to access each other's data repository as if

they were accessing their own [Lee and Zomaya, 2006]. As a result, the data in all repositories in a site can be considered as a single repository placed on the site.

In the proposed parallelization method for *data intensive bag of tasks*, Lee and Zomaya [2006] have paid more attention to taking the advantage of data sharing patterns of the tasks. Their solutions has two major phases. Firstly, the tasks are grouped based on the data shared among them. This grouping phase intends to reduce the data transfer to each site by assigning tasks, which share similar data, into one site. Secondly, the grouped tasks are dispatched to each site. Once a group of tasks is finished by a site, remaining tasks of another group will be assigned to the site. The scheduling in the second phase is dynamically handled simultaneously to the task execution. The tasks listings are also organized dynamically and task replications are used to overcome the problems arising from the dynamic nature of the grid systems.

Giersch et al. [2004] analysed the parallelization of tasks sharing files on heterogeneous master-slave platforms which form a distributed memory system. The tasks depend upon files which are initially stored in the master node and later distributed to the slave nodes. The tasks share input files. The scheduling problem is to select which file is sent to which slave node and in which order, so as to minimize the total execution time. The files are sent to the slave nodes accordingly. Giersch et al. [2004] concentrate on the scenario where files and tasks are uniform, but the slave nodes are heterogeneous. Giersch et al. [2004] prove that this scheduling problem to be NP-complete and have proposed a series of heuristics called “max-min”, “sufferage”, etc. to solve the problem. Each heuristic function selects which task is executed next in which slave node. The heuristics have been experimentally verified for good performance.

Wu et al. [2009] addresses the parallelization of tasks sharing files in the same way as Giersch et al. [2004]. In addition, they also take the data storage limitations in the nodes into account. However, their work only addresses the first step in solving

the parallelization problem, which is data distribution to a single node. As a result, they have not proposed any complete strategies to solve the scheduling problem for parallelization of tasks sharing files in limited memory.

2.4.3 Summary of Parallelization of All-to-All Comparison

In this section we have discussed many solutions proposed for parallelization of DIAC in various platforms. Among those solutions, only Wu et al. [2009] have addressed the memory capacity limitations in the computational nodes. Unfortunately, they have not proposed solutions beyond the point of data distribution to the computational nodes. Nevertheless, the research by Wu et al. is by far the most closely related to our scheduling problem.

As stated by Geng et al. [2010], only a small number of researches related to parallelization were carried out targeting multi-core systems in the past. Only Katoh and Toh [2010] have targeted the shared-memory platform among all work we have discussed. Since their problem is not data intensive and not bound by an upper limit in memory, the parallelization is fairly straightforward.

Much research [Date et al., 1993, Giersch et al., 2004, Kleinjung et al., 2002, Lee and Zomaya, 2006] for parallelization of *data intensive bag of tasks* has targeted distributed memory systems. Shared-memory systems have received less attention in this research. In our understanding, one reason for this choice is the lack of popularity of shared-memory computers until recently. Other reason is the lack of availability of powerful shared-memory systems until recently.

The parallelization strategies used for distributed-memory systems are generally different from strategies used in shared-memory systems. Therefore, much of the research discussed in this section is not directly applicable or comparable to the solutions pro-

posed for our memory management problem. To the best of our knowledge, reports do not exist in the literature to address a similar problem to ours on memory-constrained DIAC in shared-memory platforms.

2.5 Analysis of Popular Parallelization Strategies

Application parallelization is a vast research area. Various strategies have been proposed based on many factors, such as the nature of the problem, platform and targeted objective. This section will briefly discuss some of these strategies, aiming at the parallelization of DIAC in shared-memory multi-core platforms. The relationship of the methods discussed in this section to our DIAC parallelization problem is also discussed very briefly. More details about parallelization of our DIAC problem is discussed in Chapter 4.

The objectives of parallelization are not always the same. In many real-time systems [Ramamritham et al., 1990], the objective is to meet the task deadlines. In many other applications including our scheduling problem, the objective is to increase the speed of the application (decrease the total runtime) by parallelization. Therefore, this section will focus on strategies for increasing the speed of an application.

To utilize the full power of a multi-core system, it is important to evenly distribute the load among the cores [Geng et al., 2010]. Efficient load-balancing is the key to reducing the total runtime of an application in multi-core systems. Ullman [1975] has stated that "Load-balancing is a classic combinatorial optimization problem and is an NP-complete problem as difficult as the Hamilton problem" [Geng et al., 2010].

Scheduling algorithms can be divided into two main categories as static [Braun et al., 2001] and dynamic [Amalarethinam and Mary, 2011, Geng et al., 2010, Lee and

Zomaya, 2006]. Static scheduling algorithms operate based on the prior knowledge of the tasks and environment. They do not react well to dynamic load imbalances that might occur in the runtime. In comparison with dynamic approaches, less scheduling overhead in the runtime is one of the biggest advantages of static scheduling.

Dynamic scheduling takes decisions in the runtime based on both the knowledge available during the runtime and prior to the runtime. The flexibility in the runtime is one of the biggest advantages of using dynamic scheduling. However, due to the necessity of collecting, storing and analysing of state information, dynamic scheduling causes more overhead in the runtime than static approaches [Geng et al., 2010]. In our DIAC problem, prior knowledge of the execution times of tasks is limited and a dynamic approach would be the suitable path to take. However, the full awareness of data sharing patterns among the tasks makes it a good candidate for static scheduling as well.

In dynamic scheduling, the scheduling phase can be interleaved or overlapped [Hamidzadeh and Lilja, 1996]. When the scheduling phase is interleaved, working processors may have to wait until the scheduling phase finishes to receive newly assigned tasks. When the scheduling phase is overlapped, a dedicated processor can be assigned for scheduling [Amalarethinam and Mary, 2011, Hamidzadeh and Lilja, 1996]. Otherwise, the scheduling process can be distributed among processors that will take turns for scheduling tasks when they become idle.

In many scheduling problems, the tasks are independent from each other [Date et al., 1993, Giersch et al., 2004, Hamidzadeh and Lilja, 1996, Kleinjung et al., 2002, Lee and Zomaya, 2006]. However, when the tasks are dependent on each other, the Directed Acyclic Graph (DAG) based solutions have received more attention in recent research [Abdelkader and Omara, 2012, Amalarethinam and Mary, 2011, Meng et al., 2013]. Firstly, the dependencies are presented in a DAG and then the scheduling of tasks is performed based on the DAG. Our attempts to present a DIAC problem in a DAG and its evaluation will be presented in Chapter 4.

In dynamic scheduling, when a processor is not getting sufficient work to keep it busy, an event must trigger to balance the load. This event may be triggered at the processor that is becoming less loaded or idle and the processor can start looking for work. Alternatively, the scheduler may see that the load assigned to a processor is getting low, and thus assign more work to that processor [Amalarethinam and Mary, 2011, Date et al., 1993, Hill et al., 2008]. In some cases, highly loaded processors may migrate work to less loaded processors [Geng et al., 2010].

There are many dynamic scheduling strategies found in the literature. Among them, there are several common strategies standing out and used by many researchers. Most of these strategies are described in detail by Amalarethinam and Mary [2010]. Using a heuristic function to select which task to execute next in which processor is a commonly found strategy [Berman et al., 1999, Casanova et al., 2000, Giersch et al., 2004, Ramamritham et al., 1990, Xiangbin and Shiliang, 2003]. Self Adjusting Dynamic Scheduling (SADS) family algorithms are another commonly found strategy, which is based on the branch and bound search algorithms [Hamidzadeh et al., 2000, Hamidzadeh and Lilja, 1996]. Directed Acyclic Graph (DAG) based scheduling approaches have become popular for computing tasks with dependencies. In addition, there are some applications which use genetic algorithms [Omara and Arafa, 2010].

Heuristic functions used for scheduling are mainly developed to target a specific problem. Based on the current state of the system, a heuristic function will provide a representative value to help the scheduler to decide the task-processor affinity or next action. It is usually developed to represent a combination of one or more ideas affecting the objective (e.g. speed). For example, the “Max-min” heuristic developed by Giersch et al. [2004] selects the best task as the one whose objective function, on its most favourable processor, is the largest. The idea behind this heuristic is that a long task scheduled at the end would delay the end of the whole execution [Giersch et al., 2004]. Many existing heuristics are not applicable to our DIAC problem, due to

the memory capacity constraints. We developed a heuristic function based approach to evaluate the effectiveness of the technique and it will be presented in Chapter 4.

Self Adjusting Dynamic Scheduling (SADS) family algorithms are based on the branch and bound strategy [Lawler and Wood, 1966]. In the SADS algorithms, a task processor assignments are built as a tree and each path in the tree represents a schedule. Each node of the tree has a time-stamp and assigns a task to a processor. A separate processor works on building this tree of schedules based on the estimated execution times of the tasks. The process for building schedules continues through all possible branches of the search tree bound by specific constraints. In contrast to in the branch and bound algorithms, the tree is not built until a complete schedule is produced. The schedule building phase stops as soon as the least loaded processor becomes idle. Then the best partial schedule developed so far will be selected and the tasks are assigned to the processors according the partial schedule. The processors follow the schedule built so far while the schedule building process continue to run concurrently. The schedule building process continue from the end point of the previously selected schedule and continue until the next stop point. The process is repeated until all tasks are assigned to the processors.

We adapted the SADS algorithm to solve our scheduling problem. The adaptation process and the evaluation of the technique will be discussed in Chapter 4.

2.6 Popular Parallel Computing Platforms

As seen in Section 2.5, the parallelization of DIAC has been addressed on various platforms. Understanding specific properties and structure of these platforms can greatly help to distinguish the platform specific strategies for parallelization. Therefore, this section provides an overview of popular parallel computing platforms. A special at-

tention is paid to *shared memory multi-core* platforms, which is the target platform of this thesis.

A taxonomy of different computer architectures are shown in Figure 2.2. The broad classification of parallel computer models, according to Flynn [1972], is *Single Instruction Multiple Data* (SIMD) and *Multiple Instructions Multiple Data* (MIMD). SIMD systems were widely used in early days of parallel computing, but are now facing extinction. These systems consist of a large number (even thousands) of processors and each processor has a local memory. Every processor must execute the same instruction over different data at each computing or 'clock' cycle. The complexity and often the inflexibility of these systems has restricted their use mainly to special purpose applications [Trelles, 2011]. In MIMD computers, each processor can execute asynchronously and independently from other processors at its own speed on different data. This flexibility has brought more attention to high performance parallel computing to MIMD systems. In addition, MIMD computing systems are more amendable to bioinformatics [Trelles et al., 1998].

Until recently, the execution speed of all programs in general kept increasing in uniprocessor systems as the number of the transistors in the processor kept increasing. After 2004, the execution speed of single-threaded applications (also know as sequential applications) did not increase at a rate as it did formerly, even though the number of transistors in the processors was still increasing at the same rate. This situation occurred due to two predominant factors. Firstly, the processor designers were unable to increase the clock frequency of the processors at a same rate as it was earlier without exceeding the thermal and power constraints. Clock frequency is a key factor directly related to the execution time of an application. Secondly, even with the constant increase of the number of transistors in a processor, processor designers were unable to innovate new architectural designs to speed up the processor without overstepping the thermal and power constraints [Bridges, 2008].

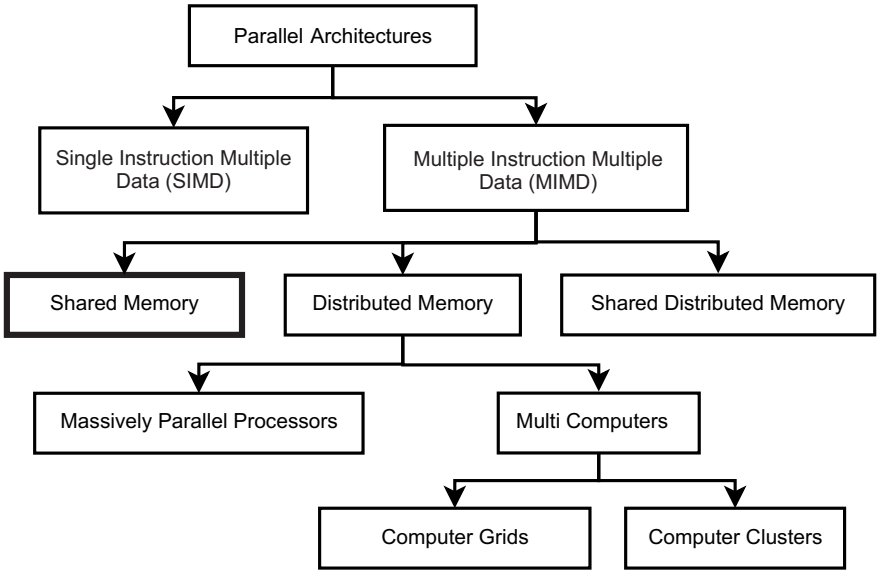


Figure 2.2: Summarized parallel computer architecture taxonomy and memory models [Trelles, 2011]

As a solution for overcoming the limitations of increasing the clock speed, the popularity of multi-core processors has increased rapidly in the recent years. At the present time, most of the desktop or laptop computers are built with multi-core processors. This popularity and availability has led our research to target the shared-memory, multi-core platforms. The cores in a multi-core processor, can execute different instructions simultaneously on different data and they are categorized in MIMDs (Multiple Instruction Multiple Data). The cores usually share the same main memory and all cores has direct access to a shared main memory.

Multi-core computers are categorized into two main categories as homogeneous and heterogeneous [Geng et al., 2010]. In homogeneous multi-core systems, all cores have the same structure and properties such as clock speed. In heterogeneous multi-core systems, cores have different structures. Usually these systems have a high-performance core and numerous general cores [Geng et al., 2010]. Due to the wide availability of homogeneous multi-core systems, our research targets homogeneous multi-core plat-

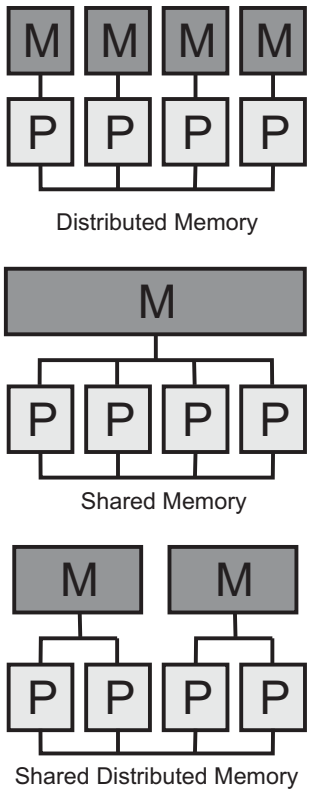


Figure 2.3: A summary of memory models in parallel computers [Trelles, 2011]

forms only.

Multi-core computers fall into the shared-memory model. Figure 2.3 shows different types of memory models used in parallel computing systems. A system is categorized as *shared-memory architecture* if any process, running in any processor, has direct access to any local or remote memory in the whole system. Otherwise, the system is categorized as distributed memory architecture [Trelles, 2011]. Most general-use multi-core computers have shared-memory architecture and are MIMDs.

Much of the research discussed in Section 2.4.1 [Date et al., 1993, Giersch et al., 2004, Kleinjung et al., 2002, Lee and Zomaya, 2006] used distributed-memory systems as the target computing platform. A major advantage of the shared-memory architec-

ture over the distributed-memory architecture is that the processors or cores can share data in the memory, which is generally much slower in distributed-memory systems. In distributed memory architectures, each processor has its own local memory (see Figure 2.3). A processor does not have direct access to any of the memories other than its own. Each node of the system has a processor and a dedicated memory. The nodes communicate through an external network. These external links are significantly slower than the direct memory access (e.g. Ethernet based networks). In some systems, specially designed high speed connections such as InfiniBand [Yu et al., 2006] are used to link the nodes to speed up the links between nodes. Therefore, minimizing internode communication is one of the major concerns in developing parallelization strategies for distributed memory systems. As a result, parallelization strategies designed for distributed memory systems are typically twofold as data distribution and task distribution.

Distributed memory systems are usually found in two different categories as clusters and grids. In general, a cluster of computers is a closely coupled set of computers and a computer grid is a loosely coupled network of computers which might expand over a topologically wide area. An example of a cluster of computers is a set of computers in a High Performance Computing (HPC) laboratory where all computer nodes are connected to the same Local Aria Network (LAN), usually through high-throughput connections [Giersch et al., 2004]. An example for a grid is a system which utilizes all computers across a university network to solve computationally intensive jobs [Lee and Zomaya, 2006, Moretti et al., 2010].

In addition to shared and distributed memory models, a new model called shared-distributed memory is also emerging (see Figure 2.3). In this model, a set of shared-memory computers with multiple processors or cores are connected by an external network which is similar to a distributed-memory system. At the present time, many cluster of computers can be categorised in to this model since the nodes of the clusters

are usually multi-core shared-memory computers.

2.7 Simulators for Parallel Algorithms

Simulation of parallel algorithms is an important part of designing a parallel algorithm through a rigorous research process. Analysis of the behaviour of an algorithm and performance evaluations can be performed through a simulator. Higher execution speed is one of the most important features of such simulators. This section discusses existing simulation tools and their suitability for our research on the DIAC problem.

There are three categories of simulators to study the characteristics of parallel programs [Sudhakar, 2006]:

1. execution driven simulators,
2. trace driven simulators; and
3. event driven simulators.

Execution driven simulators are commonly found in the literature [Austin et al., 2002, Ceze et al., 2003, Magnusson et al., 2002, Prakash and Bagrodia, 1998, Zhi et al., 2010]. They run the actual program code after a preprocessing phase (i.e. a compiling process), in a simulated parallel hardware environment similar to a virtual machine. In the process instrumentation code is injected to the program to measure parameters, such as the timing, count, frequency and type of instructions [Sudhakar, 2006]. Therefore, the compiled machine code of the simulated program is executed at some point to simulate the behaviour of the application. The execution time of a simulated task depends on the code used to mimic the task or the actual code of the task. Therefore, modifications are usually difficult to make for the runtime of the tasks.

Trace driven simulators [Mckinley and Trefftz, 1993, Reinhardt et al., 1993] are mainly

useful for studying the performance characteristics of the cache coherence and memory consistency protocols in a shared memory system [Sudhakar, 2006]. Hence they do not meet our requirements in simulating the DIAC problem.

Event driven simulators [Pertel, 1992, Sudhakar, 2006] capture the special events such as massage passing, I/O events and use these events in the simulation process. This type of simulators better meet our requirements for the DIAC problem. However, most of the existing simulators involve a code generation process to analyse a code, which may be annotated, and generate the simulator code. They still do not take the advantage of non-pre-emptive tasks and are usually computationally heavy in the runtime. In addition, these simulators still use methods like counting instructions to determine the execution time of the tasks. Therefore, modifying the task execution times based on imagined or harvested values is complicated and can be inaccurate.

A list of expectations or requirements in our research from a simulator is given below.

1. Theoretical analysis and evaluation of the performance of parallel algorithms. We should be able to emulate scenarios such as zero execution time for scheduling.
2. Programmatical and easy change of exact execution times of *loads* and comparisons. We should be able to experiment with different scenarios with variations of execution time of tasks.
3. Ability to simulate various number of cores and parallel read disks.
4. Being sufficiently lightweight and fast enough for use as a simulator for the optimum parameter prediction in the algorithms (especially in the real-runtime when parallel disk reads are feasible).
5. Ability to use simulations in conjunction with other applications such as the application developed for Simulated Annealing.
6. Ability to reproduce the same scenario or behaviour over and over again for problem identification, validation and improvements of the algorithms (deterministic

simulator).

7. The tasks are expected to have continuing and uninterrupted affiliation with the processors until it is completed. Therefore, the simulator must be able to exploit the performance gain by jumping from one CPU clock cycle to the next useful cycle quickly.

The above discussion of the existing simulators makes it clear that some of the above expectations are beyond what we can achieve from the existing simulators. Since our objective is to design a scheduling algorithm for the DIAC problem, in the design process and behavioural analysis process, our interest mainly lies in the theoretical behaviour of the algorithm in a controlled environment. Utilizing existing systems in general, which are based on instruction level simulations, poses a complicated process and does not guarantee many of the above mentioned expectations or requirements. A part of the optimum parameter selection process, which will be discussed in Section 5.4, is based on simulations.

Chapter 7 of this thesis will develop a novel and flexible simulation environment specific to the context of our research. This simulator is flexible and can be easily adapted by other research with similar expectations or requirements. Not only our research but also much other recent work [Amalarethinam and Mary, 2011, Giersch et al., 2003, 2004, Wu et al., 2009] done in a similar context use specialized custom simulators designed to suit their requirements in the experiments.

2.8 Summary of the Literature Review

The DIAC is an important and frequently found problem in data intensive bioinformatics and other applications. The composition vector (CV) method is one of the many applications which use DIAC. Our focus on the features of the CV method does not

influence the generality of the solutions developed by us for parallelization of DIAC.

The DIAC can be categorized in the *data intensive bag of tasks* (*DBoT*) class of application. Parallelization of *DBoT* has received more attention in recent researches.

Although there is much research on parallelization of DIAC, limited research exists for shared-memory multi-core systems. Not only for DIAC, overall research targeting parallelization in shared-memory multi-core systems is limited compared to the distributed-memory systems. To the best of our knowledge, research reports have not been found on the research driven towards parallelization of DIAC in shared-memory systems with limited memory. However, there are methods and strategies which can be adopted with improvements to solve the DIAC parallelization problem.

At the present time, the shared-memory multi-core systems and shared-distributed-memory systems have become popular and widely available. New parallelization strategies need to be developed specifically targeting these systems. Since our research targets DIAC in shared-memory multi-core platforms, many applications can benefit from the proposed solutions.

Chapter 3

Memory Management in Uni-processor Systems

As the first step of parallelization of *Data Intensive All-to-all Comparison (DIAC)* in limited memory, we isolate the memory management problem. This chapter answers research question 1 specified in Section 1.3. The question is how memory should be managed for DIAC in single-core (uni-processor) platforms to minimize the time spent on *load* operations. This chapter focuses only on memory management in single-core (uni-processor) systems without considering the parallelization. If the memory is assumed to be unlimited, there is no need for memory management since comparisons can start after loading all data items to the memory, as performed in Algorithm 1. Therefore, only the situation where memory is insufficient to hold all data items (i.e. preprocessed items) is addressed in this chapter.

In limited memory, data items must be swapped in and out from the memory to complete all comparisons. The number of *loads* used to complete all comparisons largely depends on the sequence of loading them to, and unloading them from the memory. In this process, if more redundant *loads* (i.e. *loads* which are over the minimally required

loads to complete a DIAC) are performed, the runtime may unnecessarily increase.

The aim of this chapter is to develop heuristics which run fast and minimize the number of *loads* when memory is insufficient for holding all data items at the same time. The algorithms are developed bearing in mind the scalability on different computer platforms with different configurations.

The chapter is organized as follows. First, we state and model the memory management problem. Then novel heuristics are developed and optimum parameters for the proposed algorithms are proven. Next a close-fitting theoretical lower bound for the minimum number of *loads* required for a DIAC to be completed under memory constraints is derived. Finally, the performance of the heuristics is validated experimentally in comparison to other existing methods. A significant portion of this chapter has been accepted as a journal publication [Krishnajith et al., 2014].

The main contributions of this chapter are:

1. A novel scalable memory management algorithm which minimizes the number of *loads* for DIAC.
2. An analysis of different possible algorithms as well as theoretically selecting optimal configuration parameters for the proposed algorithm.
3. A close-fitting theoretical lower bound for the minimum number of *loads* required for a DIAC to be completed under memory constraints.

3.1 Problem Formalization and Strategy Formulation

The *load* process of a data item is computationally intensive. In limited memory, multiple *loads* of each data item are required to complete all comparisons. Therefore, by intelligently loading data items in such a way that the number of *loads* is minimized

will increase the speed of the DIAC significantly.

It is usually faster to read a preprocessed data item from the disk rather than preprocessing it every time when loaded to the memory [Yu et al., 2010b]. Since there is no reliable method to estimate the size of the data items before they are preprocessed (for example, see Appendix A), for accurate memory management, the preprocessed size of every data item must be known. Therefore, if the size of the preprocessed data items are different from the original data items, each data item must be preprocessed at least once before starting the comparisons. The preprocessed data items can be written to the disk at this stage, for faster retrieval in later *loads*. This has already been discussed in details in Chapter 1 and Section 2.1. As a result, we assume that the memory requirement of each data item is known.

Let the maximum number of data items which can be held in the memory be M and the number of data items to be compared be N. The set of input data items are denoted by G_i where $0 \leq i < N$. Since the correlation matrix results from a DIAC is symmetric, only $N(N-1)/2$ comparisons are required to complete a DIAC, resulting in a triangular region of pair-wise distances.

There is a condition in order to complete a correlation matrix calculation: $\forall i, j \in [0, N); i < j$, G_i and G_j must be present in the memory together at least once throughout the calculation. Following this condition, we propose following conceptual procedure to complete all comparisons in a DIAC:

1. Load an initial set of M data items into memory.
2. Compare all pairs of data items that are currently in memory that have not already been compared.
3. Decide which set of items to *load* into memory and which set of items to unload (to make way for the new items).
4. Return to Step 2 until all comparisons are complete.

By using the above procedure, an algorithm will be progressively developed in the next few sections for scalable computing with efficient memory utilization. In the process of developing the algorithm, an intermediate algorithm will be presented first which can only handle uniformly sized data items. Then the intermediate algorithm is extended to the final algorithm which is capable of handling non-uniformly sized data items and practical problems such as fluctuations in free memory. Please note that simple strategies such as blocking the correlation matrix into rectangular regions designed to fit into memory do not by themselves lead to optimal results.

3.2 Algorithm Development

In considering which sets of sequences to *load* and unload in the above general procedure, we have a combinatorial number of possibilities to consider (over the lifetime of the DIAC calculation). To reduce the number of possibilities we use a heuristic approach in which the set of data items in memory are divided into two subsets set A and set B. Once we have compared all pairs of sequences currently in memory (i.e. $A \cup B$), we keep the subset set A in memory, unload the current set in set B, and *load* in a new set B. We continue this until we have compared the items in set A with all other items; we then *load* a new set A. So, set A is the set that we choose to keep in memory for a longer term, while the set B is rapidly swapped in and out.

3.2.1 Algorithm

The data items are indexed from 0 to $N-1$. Thus, set A and set B will be represented as contiguous ranges of indexes in our algorithms. They have the sizes of α and β, respectively. The proposed intermediate algorithm is depicted in Algorithm 4 and numerically illustrated in Figure 3.1.

Algorithm 4 General algorithm to calculate all comparisons

1: **procedure** COMPAREALL(N, M, α, β) // α and β are sizes of set A and set B
2: initialize **global** $matrix[N][N]$ with -1;
3: $B \leftarrow \{G_x : x \in \mathbb{Z}, 0 \leq x < \beta\}$
4: LOAD data items in set B
5: Set flag $forward = true$
6: Set A to be empty
7: **for** $p = 0 \rightarrow N - \beta - \alpha$ **step** α **do**
8: UNLOAD all data items in set A
9: $A \leftarrow \{G_x : x \in \mathbb{Z}, p \leq x < p + \alpha\}$
10: **if** $forward == true$ **then**
11: $A' = A - B$ // Data items in A and not in B
12: LOAD data items in set A'
13: **for** $i = p + \alpha \rightarrow N - 1$ **step** β **do**
14: $B \leftarrow \{G_x : x \in \mathbb{Z}, Min(i, N - \beta) \leq x < Min(i + \beta, N)\}$
15: **if** $i > p + \alpha$ **then**
16: $B' \leftarrow \{G_x : x \in \mathbb{Z}, i - \beta \leq x < Min(i, N - \beta)\}$
17: UNLOAD all data items in set B'
18: **end if**
19: $B'' \leftarrow \{G_x : x \in \mathbb{Z}, Max(i, p + \beta) \leq x < Min(i + \beta, N)\}$
20: LOAD all data items in set B''
21: COMPAREALL (A, B)
22: **end for**
23: **else**
24: LOAD all data items in set A
25: **for** $i = N \rightarrow p + \alpha + 1$ **step** β **do**
26: **if** $i < N$ **then**
27: $B \leftarrow \{G_x : x \in \mathbb{Z}, Max(i - \beta, p + \alpha) \leq x < Max(i, p + \alpha + \beta)\}$
28: $B' \leftarrow \{G_x : x \in \mathbb{Z}, Max(i, p + \alpha + \beta) \leq x < i + \beta\}$
29: UNLOAD all data items in set B'
30: $B'' \leftarrow \{G_x : x \in \mathbb{Z}, Max(i - \beta, p + \alpha) \leq x < i\}$
31: LOAD all data items in set B''
32: **end if**
33: COMPAREALL (A, B)
34: **end for**
35: $forward =!forward$ // Reverse forward flag

Continued on the next page...

```
36:         end if
37:         q = p
38:     end for
39:     UNLOAD all data items in set A
40:     if forward == true then
41:         A' ← {G_x : x ∈ ℤ, p + β ≤ x < N}  // p is it's last value in the for loop
42:     else
43:         A' ← {G_x : x ∈ ℤ, p ≤ x < N − β}
44:     end if
45:     LOAD all data items in set A'
46:     COMPAREALL (A', B)
47: end procedure
48: procedure COMPAREALL(A, B)
49:     COMPARE (x, y) ∀x, y where x, y ∈ A ∪ B
50: end procedure
51: procedure COMPARE(i, j)
52:     if matrix[i][j] == −1 then
53:         matrix[i][j] = matrix[j][i] = COMPARISONFUNCTION (G_i, G_j)
54:     end if
55: end procedure
```

It is seen from Algorithm 4 that the set B initially sweeps from left to right across the columns of the matrix. However, once we reach the right edge of the matrix (and need to load a new set A), rather than unloading set B and moving back to the start of the next row, we instead keep set B in memory and reuse it for the next set A. Set B then sweeps backwards from right to left across the next row. In line 8, the previous set A is unloaded from memory. Set A is empty in the first iteration. Lines 12 or 24 then loads the next set A. However, if the previous iteration has moved set B backwards (i.e., $forward = true$), a subset of the new set A will already be in memory because of the 'bringing forward' of set B. Therefore only a subset A' of set A (that is not already loaded) needs to be loaded in line 12.

The loops starting at lines 13 and 25 move set B forward and backward, respectively. Within these loops, lines 17 and 29 unload the previous set B. Furthermore, lines 20 and 31 load the next set B. The inner loops for set B terminate after they get within β (the size of set B) from the (left or right) end of the row. If the remainder for set B

in the last inner loop iteration is less than β, it will be slid, so that a maximally sized set B is preserved for the next iteration. To conduct this sliding, when set B is at a boundary, only a subset of previous set B, B', is unloaded at lines 17 and 29. Also, when this sliding happens only a subset of set B, B'', is loaded at lines 20 and 31.

3.2.2 Graphical Illustration of the Algorithm

An illustration of our algorithm is shown in Figure 3.1, where N, the number of data items, is 14; M, the maximum number of data items that can be held in the memory, is 5; and the sizes of set A and set B are $\alpha = 3$ and $\beta = 2$, respectively. The data items are indexed from 0 to 13. In Figure 3.1, each square represents a comparison between the data items corresponding to the column and row numbers. The number inside a square represents the iteration number of the loop starting at line 7 in Algorithm 4.

3.2.3 Illustrative Example of the Algorithm

In the following, Algorithm 4 is further illustrated using the example in Figure 3.1.

- At line 4, an initial set B ($\{G_0, G_1\}$) is loaded. Set A for the first iteration is $\{G_0, G_1, G_2\}$.
- Since the current set B, which is a subset of set A, is already in memory, only $A - B$ ($\{G_2\}$) is loaded at line 12.
- At this stage, set B should be moved forward because the *forward* flag is *true*.
- The first set B is loaded adjacent to set A. The first five sets loaded to set B in the loop starting at line 13 are $\{G_3, G_4\}$, $\{G_5, G_6\}$, $\{G_7, G_8\}$, $\{G_9, G_{10}\}$ and $\{G_{11}, G_{12}\}$.
- In the loop starting at line 13, each set B is loaded and then all uncompleted comparison with the current set A and set B are completed at line 21.

- After the first five sets in set B, the remaining data items are insufficient for a full set B. Therefore, instead of loading a full set B, the remainder is loaded and a number of data items equivalent to the remainder are unloaded from the beginning of the current set B. In this case the number remaining is 1; and G_{11} is unloaded and G_{13} is loaded. Thereafter, set B becomes $\{G_{12}, G_{13}\}$. We call this process "sliding" set B. The sliding process preserves a full set of set B for the next iteration. The sliding process is the reason to use B' instead of B at line 17 to unload set B. B' is always similar to the previous set B, unless sliding is in place.
- Similarly, B'' at line 20 which loads the next set B, is always similar to the next set B unless sliding is in place. Importantly, the last set in set B is not unloaded in both inner loops.
- For the next iteration, the *forward* flag is set to *false* at line 35.
- Set A is stepped by α and a full set A is loaded at line 24 which is $\{G_3, G_4, G_5\}$.
- The loop starting at line 25 moves set B backward (since *forward* is *false*) and for each set B all uncompleted comparison with the current set A and set B are completed at line 33. The next four sets of set B are $\{G_{12}, G_{13}\}, \{G_{10}, G_{11}\}$, $\{G_8, G_9\}$ and $\{G_6, G_7\}$.
- It is noted that the first set B ($\{G_{12}, G_{13}\}$) is already in the memory from the previous iteration. The "if" condition at line 26 avoids loading this set.
- If the remainder for the last set B is less than β (this is not the case here), the set B is slid ('sliding' process) by unloading the last data items at the end of the set B and loading the remainder to the front. Note again that the last set in set B is not unloaded. This last set in set B ($\{G_6, G_7\}$) will be a subset of next set A (G_6, G_7, G_8) and will be excluded from loading at 13 $(A - B)$ in the next iteration.
- The loop at line 7 is repeated until the remainder of the data items after the latest set A is less than M. At lines 40-45, if the remainder is not zero, the remainder is loaded to set A excluding the current set B which was preserved from the last

iteration.

- Then, all uncompleted comparisons with the current set A and set B are completed at line 46.

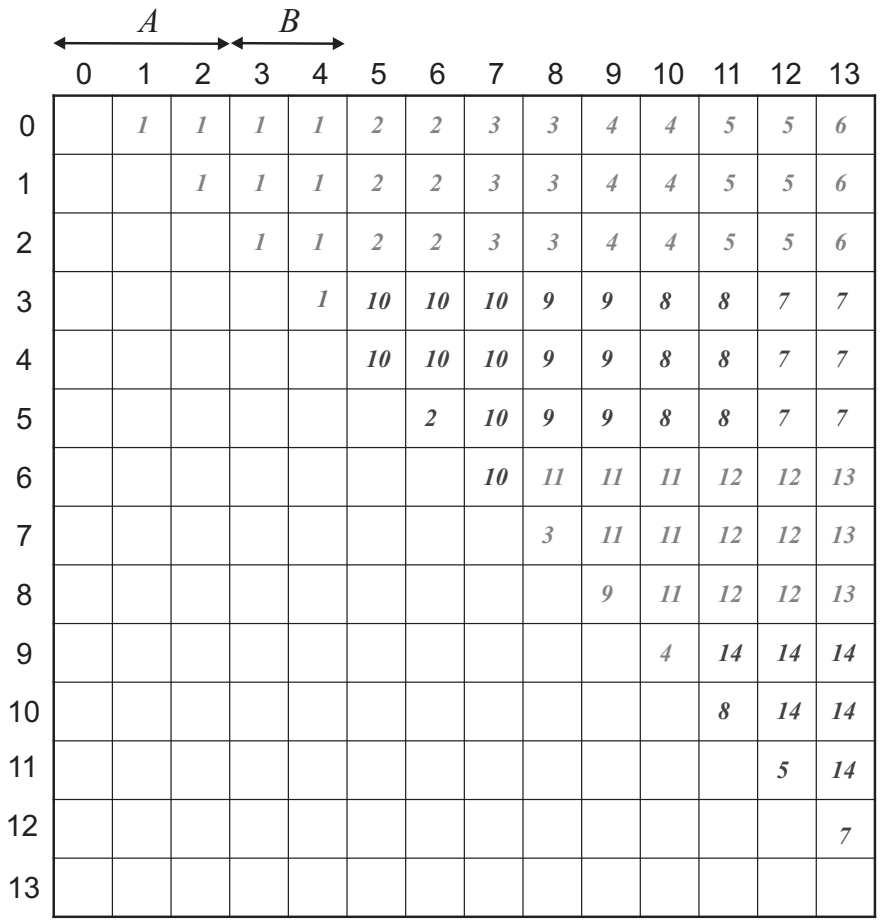


Figure 3.1: The comparisons completed in each iteration in Algorithm 4. The number at each square shows the iteration number. This depiction shows a comparison of 14 data items, assuming that a maximum of 5 data items can be held in the memory. The sizes of the sliding sets A and B are $\alpha = 3$ and $\beta = 2$ respectively.

3.2.4 Discussion on Algorithm 4

Algorithm 4 does fewer loads than the algorithm proposed by Mueen et al. [2010] (depicted in Algorithm 3). The main improvement in our approach is the 'bringing forward' of set B from one iteration to another thus saving some loads of the next iteration.

As M represents the maximum number of data items that the memory can hold at a time, it should be noted that the following condition must hold:

$$2 \leq \alpha + \beta \leq M. \tag{3.1}$$

For the full utilization of memory at any given time, we use the following condition in Section 3.3.1:

$$2 \leq \alpha + \beta = M. \tag{3.2}$$

The preprocessing phase (*load* process) in some applications [CVTree, 2011, Yu et al., 2010a] requires a significant amount of memory. We assume that the memory requirement of a *load* process does not exceed the final size of a preprocessed data item. It is important to note that M, the maximum number of data items that can be held in the memory, is decided after taking into account other memory requirements such as memory required for the program itself.

3.3 Theoretical Results

This section develops theoretical results related to Algorithm 4 and memory management in the our DIAC problem. The results are categorized in three subsections and 8 theorems. Section 3.3.1 developed theorems for deciding optimum parameter set-

tings for Algorithm 4. Section 3.3.2 derives a lower bound for the number of *loads* required to complete a DIAC under memory constraints. Section 3.3.3 presents a time complexity analysis of Algorithm 4.

3.3.1 Theorems for Algorithm 4

This subsection theoretically derives optimum parameter settings for Algorithm 4. Theorems 1 and 2 give the expression L, the number of *loads* in terms of different memory utilizations. Theorem 3 establishes the lower and upper bounds of L. Theorem 4 derives the optimal β value that gives the minimal L, for $M < \frac{2N+1}{3}$; while Theorem 5 deals with the case of $M \geq \frac{2N+1}{3}$ for the minimal L value.

Theorem 1. *When α and β are set such that the condition in Equation (3.1) is met, the number of loads L, of Algorithm 4 is:*

$$L = N(t' + 2) - \beta(t' + 1) - \frac{1}{2}\alpha(t' + 1)(t' + 2), \tag{3.3}$$

where

$$t' = \left\lfloor \frac{N - \alpha - \beta}{\alpha} \right\rfloor. \tag{3.4}$$

and $\lfloor x \rfloor$ is a floor function which rounds down x to the nearest integer.

Proof. According to the proposed algorithm (Algorithm 4) in Section 3.2, the following series can be identified to count the number of *loads*. There are $t' + 1$ iterations in the algorithm in the loop at line 7.

- For a general iteration (of the loop starting at line 7), set B is brought forward from the previous iteration, but in the first iteration no data items are in the memory, so at line 4, a new set B is loaded (β *loads*);
- When set B is stepped forward within the loop starting at line 13, $N - (p + \alpha)$ data items are loaded. If $\alpha \geq \beta$, a subset of set A is taken from the last set B of

the previous iteration and the rest (A') is loaded at line 12. Therefore, to *load* set A, only $\alpha - \beta$ loads are needed. Thus when $\alpha \geq \beta$, the number of *loads* in each iteration of the loop on line 13 is:

$$\underbrace{(N-p-\alpha)}_{\text{While Moving set } B} + \underbrace{\alpha}_{\text{For set } A} - \underbrace{\beta}_{\text{Brought Forward}} = N - p - \beta;$$

If $\alpha < \beta$, the full set A is in the memory from the last set B of the previous iteration and no *load* is done at line 12. In the loop starting at line 13, a subset of size $\beta - \alpha$ of the first set B is also already in the memory from the previous set B. Therefore, when $\alpha < \beta$, the number of *loads* in each iteration is:

$$[N - (p + \alpha) - \beta] + \alpha = N - p - \beta.$$

- When set B is stepped backward within the loop at line 25, $N-(p+\alpha)$ data items are loaded. However, since the first set B is taken from the last set B of previous iteration, the actual number of *loads* while stepping set B is $N - (p + \alpha) - \beta$. To *load* set A at line 24, α *loads* are needed. Thus the number of *loads* in the iteration is:

$$[N - (p + \alpha) - \beta] + \alpha = N - p - \beta.$$

- The iteration variable p marks the start of the current set A and its values are $\{0, \alpha, 2\alpha, \ldots, t'\alpha\}$; After $t' + 1$ iterations, if there are any remaining data items, it is insufficient for a normal iteration. So, the remaining data items are used to partially fill the current set A at line 45. The current set B which is a subset of the new set A is already in the memory. So, current set B is skipped when loading the new set A at lines 41 and 43. Thus, the number of *loads* for this section of code is: $[N - \beta - t'\alpha]$.

Therefore, we have

$$\begin{aligned} L &= \beta + \sum_{i=0}^{t'} (N - i \times \alpha - \beta) + [N - \beta - t'\alpha] \\ &= N + \sum_{i=1}^{t'} (N - i \times \alpha - \beta) - t'\alpha \\ &= N(t'+2) - \beta(t'+1) - \frac{1}{2}\alpha(t'+1)\,(t'+2)\,. \end{aligned}$$

This completes the proof. □

Following Theorem 2 gives an alternative expression of the number of loads L.

Theorem 2. *If α and β are chosen such that the Condition Equation (3.2) is met, then the number of loads L of Algorithm 4 is:*

$$L = N(t+2) - \beta(t+1) - \frac{1}{2}(M-\beta)(t+1)(t+2), \tag{3.5}$$

where

$$t = \left\lfloor \frac{N-M}{M-\beta} \right\rfloor. \tag{3.6}$$

Proof. From Equation (3.2), we have

$$\alpha = M - \beta. \tag{3.7}$$

Substituting Equation (3.7) into Equations (3.3) and (3.4) gives Equations (3.5) and (3.6), respectively. □

The following Theorem 3 establishes lower and upper bounds of the number of *loads* L of Algorithm 4.

Theorem 3. *If α and β are chosen such that Equation (3.2) is met, then the number of*

loads L of Algorithm 4 is bound by a lower-bound $\underline{L}$ and an upper-bound $\overline{L}$ i.e.,

$$\begin{aligned}
&\underline{L} \leq L \leq \overline{L} \\
&\underline{L} = N + \left(\frac{N-M}{2}\right)\left(\frac{N-\beta}{M-\beta}\right) \\
&\overline{L} = N + \left(\frac{N-M}{2}\right)\left(\frac{N-\beta}{M-\beta}\right) + \frac{1}{8}(M-\beta)
\end{aligned} \tag{3.8}$$

Proof. From Equation (3.6), we have

$$t = \left(\frac{N-M}{M-\beta}\right) - \epsilon \text{ ; where } 0 \leq \epsilon < 1 \tag{3.9}$$

Substituting Equation (3.9) to Equation (3.5) yields

$$L = N + \left(\frac{N-M}{2}\right)\left(\frac{N-\beta}{M-\beta}\right) + \underbrace{\frac{1}{2}(M-\beta)(\epsilon-\epsilon^2)}_{\text{error} = E} \tag{3.10}$$

$$E = \frac{1}{2}(M-\beta)(\epsilon-\epsilon^2)$$

$$0 \leq E \leq \frac{1}{8}(M-\beta) \tag{3.11}$$

because

$$0 \leq \epsilon - \epsilon^2 \leq \frac{1}{4}.$$

The lower and upper bounds of L in Equation (3.8) are derived by substituting E as defined in Equation (3.10) into Equation (3.11). This completes the proof. □

Following Theorem 4 establishes the value of β for minimum L when $M < \frac{2N+1}{3}$.

Theorem 4. *If α and β are chosen such that the Condition (3.2) is met, and if*

$$M < \frac{2N+1}{3}, \tag{3.12}$$

then the number of loads L of Algorithm 4 reaches its minimum L_{min} at $\beta_{min} = 1$ and

$$L_{min} = N(t+2) - (t+1) - \frac{1}{2}(M-1)(t+1)(t+2), \tag{3.13}$$

where

$$t = \left\lfloor \frac{N-M}{M-1} \right\rfloor . \tag{3.14}$$

Proof. In Equation (3.10), β is not a continuous variable, so we cannot differentiate L with respect to β. So, let k be a continuous variable (over the range $1 \ldots M-1$) that equates with β at discrete points. From Equation (3.11); we have

$$0 \leq E' \leq \frac{1}{4}$$

where $E' = \epsilon - \epsilon^2$ which is independent of k. Substituting k for β in Equation (3.10) yields

$$\frac{dL}{dk} = \frac{1}{2}\left(\frac{N-M}{M-k}\right)^2 - \frac{1}{2}E' \tag{3.15}$$

Substituting $0 \leq E' \leq \frac{1}{4}$ in Equation (3.15) gives

$$\frac{1}{2}\left(\frac{N-M}{M-k}\right)^2 - \frac{1}{8} \leq \frac{dL}{dk} \leq \frac{1}{2}\left(\frac{N-M}{M-k}\right)^2 \tag{3.16}$$

If $\frac{dL}{dk} > 0$, then L increases as k increases. If we consider only the range of k, then the right side of Equation (3.16) is always greater than zero:

$$\frac{1}{2}\left(\frac{N-M}{M-k}\right)^2 > 0.$$

If the left side of Equation (3.16) is positive:

$$\frac{1}{2}\left(\frac{N-M}{M-k}\right)^2 - \frac{1}{8} > 0. \tag{3.17}$$

then either:

$$k > 3M - 2N, \tag{3.18}$$

or

$$k < 2N - M. \tag{3.19}$$

Since $M \leq N$, Equation (3.19) is always true within the limits of k. Therefore, if Equation (3.18) is satisfied, $\frac{dL}{dk} > 0$ is satisfied within the range of k. By applying

the lowest limit of k which is 1, for Equation (3.18) to find the condition for Equation (3.18) to be satisfied, we get Equation (3.12).

Therefore, when the condition in Equation (3.12) holds, $\frac{dL}{dk} > 0$ for all the k of its range. In this case, L increases as k increases within its range, as shown in Figure 3.2. L takes its minimum L_{min} at $k = 1$. L_{min} in Equation (3.13) is achieved by substituting $\beta = 1$ into Equation (3.5) in Theorem 2. □

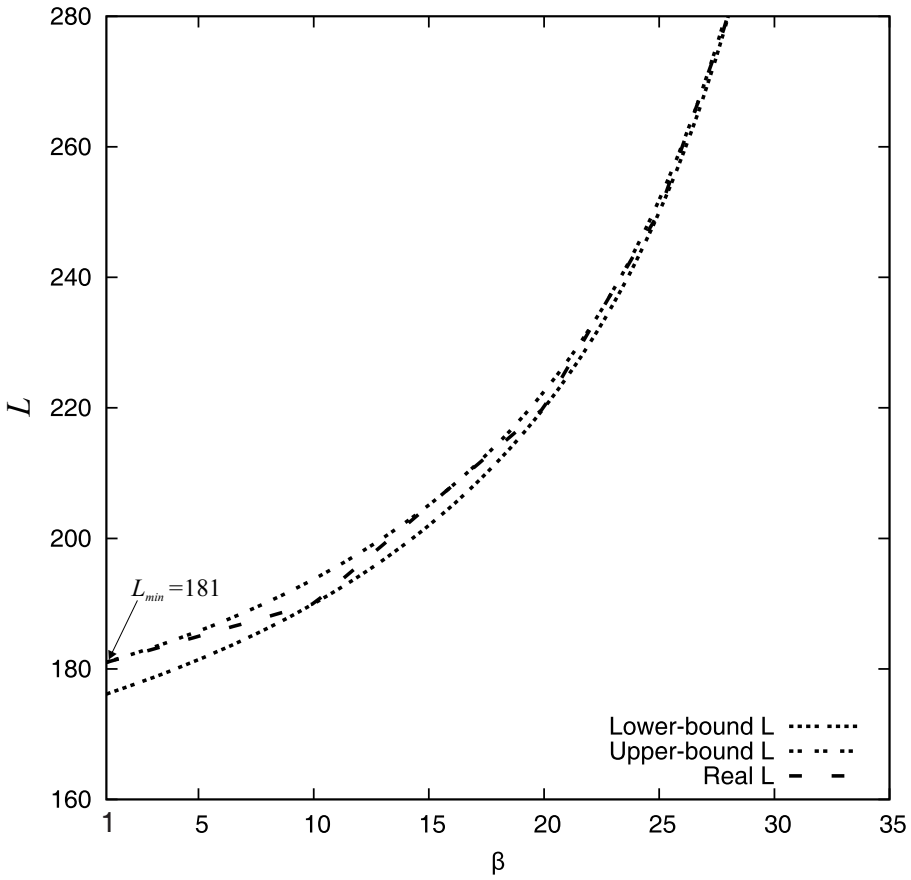


Figure 3.2: L versus β for $N = 100$ and $M = 40$. In this case $M < \frac{2N+1}{3}$ and L reaches its minimum $L_{min} = 181$ at $\beta = 1$.

Following Theorem 5 establishes the value of β for minimum L when $M \geq \frac{2N+1}{3}$.

Theorem 5. *If α and β are chosen such that Condition (3.2) is met and*

$$M \geq \frac{2N+1}{3}, \tag{3.20}$$

then the number of loads L of Algorithm 4 reaches its minimum L_{min} at a β_{min} such that

$$1 \leq \beta_{min} \leq 3M - 2N. \tag{3.21}$$

In this case, L_{min} is narrowly bound by an upper bound $\overline{L_{min}}$ and a lower bound $\underline{L_{min}}$:

$$\underline{L_{min}} \leq L_{min} \leq \overline{L_{min}} \tag{3.22}$$

where

$$\underline{L_{min}} = N + \left(\frac{N-M}{2}\right)\left(\frac{N-1}{M-1}\right)$$
$$\overline{L_{min}} = 2N - M.$$

Proof. Following the idea of the proof for Theorem 4, when Equation (3.12) is not satisfied, we need to consider the upper and lower bounds of L to find L_{min}. After substituting k for β in Equation (3.8), we have

$$\frac{d\underline{L}}{dk} = \frac{1}{2}\left(\frac{N-M}{M-k}\right)^2 \tag{3.23}$$

$$\frac{d\overline{L}}{dk} = \frac{1}{2}\left(\frac{N-M}{M-k}\right)^2 - \frac{1}{8} \tag{3.24}$$

According to Equation (3.23), $\frac{d\underline{L}}{dk} > 0$ is always satisfied for the range of k. As a result, $\min_k \underline{L}$ is at $k = 1$.

By reusing the results of Equation (3.17) to solve the inequality $\frac{d\overline{L}}{dk} > 0$ we get

$$\frac{d\overline{L}}{dk} \begin{cases} < 0; & \text{for } 1 \leq k < 3M - 2N \\ = 0; & \text{for } k = 3M - 2N \\ > 0; & \text{for } 3M - 2N < k \leq M - 1 \end{cases} \tag{3.25}$$

According to Equation (3.25), when k increases, $\overline{L}$ decreases in the range $1 \leq k < 3M - 2N$ and increases in the range $3M - 2N < k \leq M - 1$. Therefore, $\min_k \overline{L}$ is at $k = 3M - 2N$.

As proven above, under the condition of Equation (3.20), $\underline{L}$ reaches its minimum at $k = 1$, while $\overline{L}$ reaches its minimum at $k = 3M - 2N$, as shown in Figure 3.3). Therefore, L_{min} resides between $\min_k \underline{L}$ and $\min_k \overline{L}$ as shown in Equation (3.22). This completes the proof. □

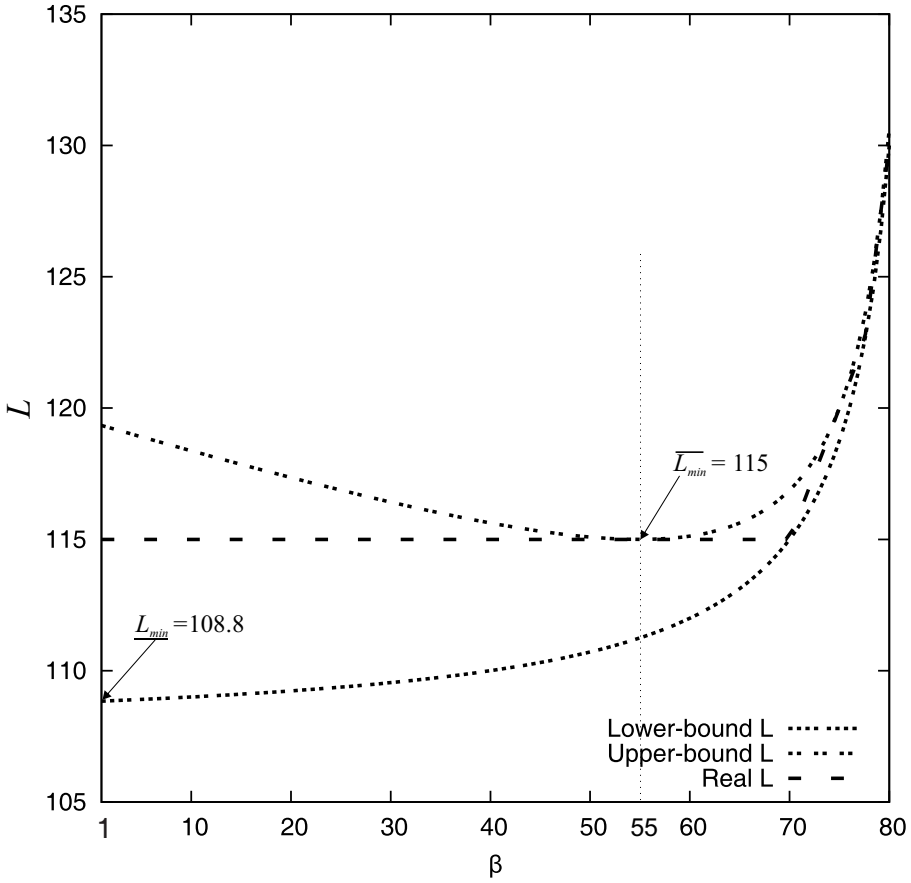


Figure 3.3: L versus β for $N = 100$ and $M = 85$. In this case $M > \frac{2N+1}{3}$. As β increases, the lower bound of L increases; while the upper bound decreases first and then increases with its minimum at $\beta = 3M - 2N = 55$. $1 \leq \beta_{min} \leq 55$ and $108.8 \leq L_{min} \leq 115$.

Since Theorem 5 only gives a range for β_{min} (i.e. value of β when L is at the minimum), a simple search technique is used to find β_{min} when $M > \frac{2N+1}{3}$. Firstly, this technique tries all β values in the range given by Theorem 5 and calculate L for each β using Theorem 2. Then the β value which produces minimum L (L_{min}) is decided as β_{min}.

In addition to the above proofs, by using a brute-force method that tries all possible combinations of *load* patterns, we have also proven that Algorithm 4 with $\beta = 1$ is one of the combinations with a minimum number of loads. Any other combination could not produce better results than the proposed algorithms up to $N = 9$ with all possible M values. Since the number of combinations grows exponentially with N, we only ran this test up to $N = 9$.

We have developed a greedy algorithm as follows: every time a new data item needs to be brought in, one data item is unloaded and one new data item is loaded such that the swap brings the most possible uncompleted comparisons. This algorithm also could not beat the number of *loads* achieved by Algorithm 4 with $\beta = 1$. We ran this test up to $N = 85$ with all possible M values.

3.3.2 Lower bound of Required Loads for All-to-All Comparison

This sections derives a close-fitting lower bound for the minimum number of *loads* required to complete an *all-to-all comparison*. The results are summarized in three theorems. Theorem 6 and Theorem 7 derive the maximum possible comparison a single *load* can bring into memory in terms of the number of data items already loaded into memory. Then Theorem 8 establishes the lower bound for the minimum number of *loads* required to complete an *all-to-all comparison*. The number of *loads* performed by Algorithm 4 with $\beta = 1$ will be compared with the derived lower bound later in this section.

Theorem 6. *If m number of data items are already loaded to memory, a new load can make available a maximum of m comparisons.*

Proof. In *all-to-all comparison* every single data item has only one comparison with every other data items and no comparison with itself. Therefore, a newly loaded item which has not completed any comparison with the already loaded items, brings m comparisons. It is the maximum number of comparisons a single load can bring. This completes the proof. □

Theorem 7. *Let N be the number of items to be all-to-all compared and M be the maximum number of data items which can be stored in memory. A single load can bring maximum of $M-1$ comparisons to memory.*

Proof. To load an item to memory, a memory slot must always be free. Therefore, if a *load* of an item is possible then a maximum of $M-1$ items can be already in memory. Therefore, maximum of $M-1$ comparisons can be bought in by a single *load* according to Theorem 6. □

Theorem 8. *At least L_{tmin} number of loads are required to complete an all-to-all comparison, where:*

$$L_{tmin} = \left\lfloor \frac{N(N-1)}{2(M-1)} - \frac{M}{2} + 1 \right\rfloor + M - 1 \tag{3.26}$$

Proof. Our objective is to find the maximum possible number of comparisons a single *load* can bring to memory at each *load*. Assuming that such a sequence of loading items exists, then we can derive the minimum number of *loads* required to complete all comparisons.

Theorem 7 proved that only a maximum of $M-1$ comparisons can be brought into memory by a single *load*. However, to bring $M-1$ comparisons to memory, there must be $M-1$ items already loaded to memory. When the memory is empty initially, clearly

there is no possibility to *load* $M-1$ items at once. The *loads* must be carried out one after another until $M-1$ items are loaded to memory before reaching the maximum possible comparisons by a single *load*, $M-1$. Until this point, Theorem 6 can be used to derive the maximum possible comparisons which can bought into memory at each stage. Thus, the following sequence gives the sum of the maximum possible comparisons that can be brought into memory, C_{init}, until the memory can be filled with $M-1$ items.

$$\begin{aligned} C_{init} &= 0+1+2+\cdots+M-2 = \sum_{i=1}^{M-2} i \\ &= \frac{(M-1)(M-2)}{2} \end{aligned} \tag{3.27}$$

There are total of $\frac{N(N-1)}{2}$ comparisons to be compared. After the first $M-1$ *loads*, there are C_{rem} number of comparisons remaining to be completed where:

$$C_{rem} = \frac{N(N-1)}{2} - C_{init}. \tag{3.28}$$

Substituting Equation (3.27) in Equation (3.28) gives,

$$C_{rem} = \frac{N(N-1)}{2} - \frac{(M-1)(M-2)}{2}. \tag{3.29}$$

After memory is filled with $M-1$ items, we assume that there exists a sequence of loading items to memory such that each *load* can bring the maximum possible number of comparisons. Therefore, thereafter each load brings $M-1$ comparisons (i.e. the maximum a *load* can bring according to Theorem 7). Since each *load* brings the maximum possible number of comparisons, there should be at least sufficient *loads* to bring all remaining comparisons (C_{rem}) to memory. Therefore, any sequence to *load* the items must be at least perform more or equal to the number of *loads* to L_{rem} where:

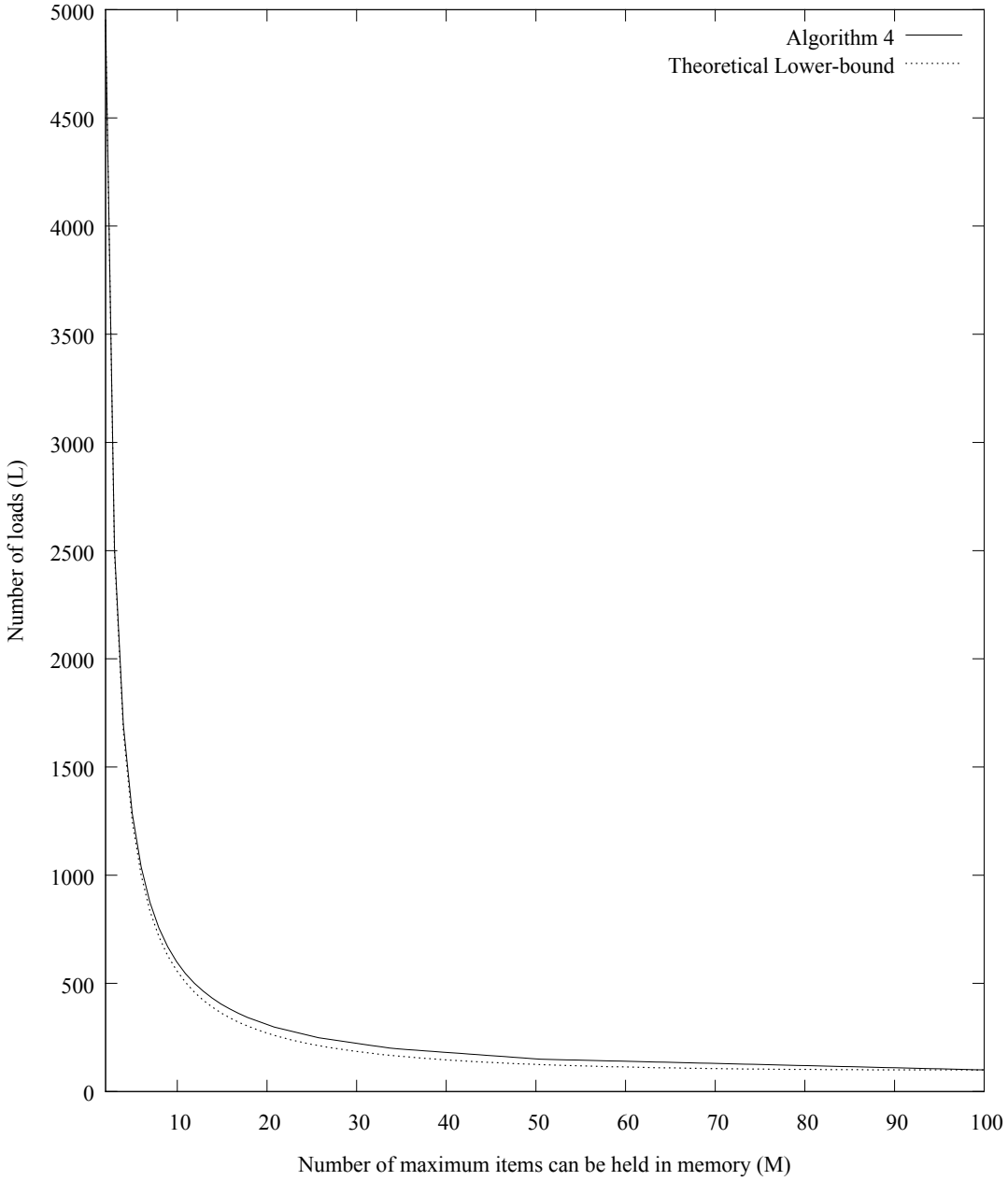


Figure 3.4: The number of *loads* of Algorithm 4 (from Equation (3.13)) and the theoretical lower bound of minimum required *loads* (from Equation (3.26)) plotted versus M for $N = 100$.

$$L_{rem} = \left\lfloor \frac{C_{rem}}{M-1} \right\rfloor . \tag{3.30}$$

To fill the memory to $M-1$ items initially, there should be a minimum of $M-1$ *loads*. Therefore, the minimum number of *loads* required to complete an *all-to-all comparison*, L_{tmin} is

$$L_{tmin} = L_{rem} + M - 1. \tag{3.31}$$

Substituting Equation (3.29) in Equation (3.30) and then substituting Equation (3.30) in Equation (3.31) gives Equation (3.26). This completes the proof. □

Figure 3.4 shows a comparison between the number of *loads* required by Algorithm 4 (with $\beta = 1$) versus the theoretically minimum required *loads* to complete an *all-to-all comparison* from Theorem 8 for all possible M values. It is seen from Figure 3.4 that the number of loads in Algorithm 4 stays very close to the theoretical lower bound of the minimum required *loads*.

The small deviation of the number of *loads* performed by Algorithm 4 from the theoretical minimum required *loads* is caused by the *loads* performed at the beginning of each row in Algorithm 4. The *loads* at the beginning of a row does not bring $M-1$ comparisons to memory with each *load*. Therefore, this small difference causes Algorithm 4 to have a slightly higher loads than the lower bound where it is assumed that there exists a such sequence that brings the maximum possible number of comparisons ($M-1$) by each *load*, after the first time the memory is filled with $M-1$ items.

3.3.3 Time Complexity of Algorithm 4

In Algorithm 4, the number of *loads* depends on the number of total data items, N and the maximum number of data items memory can hold, M. Equation (3.6) from

Theorem 2 gives L, the number of *loads* in Algorithm 4. Since L increases when M decreases, the worst case scenario for *loads* is when M is at the minimum which is $M = 2$. Therefore, the complexity of *loads* in Algorithm 4 is $\mathcal{O}(N^2)$ which is also the time complexity for *loads*. The time complexity for comparisons is also $\mathcal{O}(N^2)$ since the number of comparisons is $\frac{N(N-1)}{2}$. Therefore, the overall time complexity of Algorithm 4 when $\beta = 1$ is $\mathcal{O}(N^2)$.

3.4 Scalable Memory Management Algorithm

The theoretical results in Section 3.3.1 assumes that the maximum number of data items that can be loaded into the memory is a constant (i.e. M is a constant). This is reasonable for a set of data items which are similar in size and are loaded into a fixed size memory. However, in some applications [CVTree, 2011, Yu et al., 2010a], the size of preprocessed data items (e.g. *composition vectors* in CV method [Yu et al., 2010a]) varies considerably. Also, the available memory of a computer system may vary considerably over time due to other background processes in the system. Therefore, in practice the number of data items that can fit into memory at a time varies throughout the computation.

The results in Section 3.3.1 lead us to conjecture that $\beta = 1$ gives best results even when M is not a constant. Such a claim is difficult to formally prove given the random nature of M. We have, however, developed an extended algorithm (depicted in Algorithm 5) which addresses the challenge introduced by a variable M. In addition to assuming $\beta = 1$, following significant changes are also made in this algorithm.

Change 1: Algorithm 5 has a variable size for set A, so in the loop starting at line 21, set A is filled until the memory is only sufficient for the largest item to be loaded to set B. The algorithm is designed to leave a fixed percentage (say 10%) of

Algorithm 5 Scalable computation of correlation matrix.

```
1:  procedure CompareAll(N)
2:      for i = 0 → N − 1 do
3:          size[i] = PreprocessedSize (G_i);
4:      end for
5:      Sort data items by size[i] in descending order;
6:      p = 0; // Index of last data item in set A
7:      Set flag forward = true and step = 0;
8:      Load data item G_0
9:      while p < N − 1 do
10:         for i = p − step → p − 1 do
11:             Unload data item G_i;
12:         end for
13:         if forward then
14:             q = p + 1
15:             max_A = N;
16:         else
17:             q = p
18:             max_A = N − 1;
19:         end if
20:         Reset step = 0;
21:         while q < max_A do
22:             mem_next = size[q]
23:             if q < N − 1 then
24:                 mem_next = mem_next + size[q + 1]
25:             end if
26:             f_mem =FreeMem()−(TotalMem()∗0.1)
27:             if mem_next > f_mem then
28:                 break // Terminate the while loop
29:             end if
30:             Load data item G_q;
31:             Increment step;
32:             Increment q;
33:         end while
34:         if forward == true then
35:             for i = p + step → N − 1 do
36:                 if i ≠ p + step then
37:                     Load data item G_i;
38:                 end if
```

Continued in the next page...

```
39:                 for j = p → step + p − 1 do
40:                     COMPARE (i, j);
41:                 end for
42:                 if i ≠ N − 1 then
43:                     UNLOAD data item G_i;
44:                 end if
45:             end for
46:         else
47:             for i = N − 1 → p + step do
48:                 if i ≠ N − 1 then
49:                     LOAD data item G_i;
50:                 end if
51:                 for j = p → step + p − 1 do
52:                     COMPARE (i, j);
53:                 end for
54:                 if i ≠ p + step then
55:                     UNLOAD data item G_i;
56:                 end if
57:             end for
58:         end if
59:         for i = p → Min(step + p, N) − 1 do
60:             for j = i + 1 → Min(step + p, N) − 1 do
61:                 COMPARE (i, j);
62:             end for
63:         end for
64:         Increment i by step;
65:         forward = inverse of forward;
66:     end while
67: end procedure
68: procedure COMPARE(i, j)
69:     matrix[i][j] = matrix[j][i] = COMPARISONFUNCTION (G_i, G_j)
70: end procedure
```

the total physical memory unused, to avoid the virtual memory system slowing performance.

Change 2: The data items are sorted by size in descending order at line 5. This sorting helps to quickly calculate the amount of memory required for the largest upcoming data item for set B at line 24. It also improves the performance as described in Section 3.5.7.

Change 3: In some applications, the final size of a data item after preprocessing is hard to predict in advance (Appendix A). Therefore, if required, Algorithm 5 preprocesses and records the sizes of data items before the comparisons begin (within the loop of Algorithm 5 starting at line 2) to accurately decide the amount of memory required before a *load* is done.

When a data item is preprocessed for the first time, it can be written to the disk to avoid preprocessing again. However, reading a previously written data item from the disk is not always faster than generating it from scratch. Our experiments show that when a data item becomes larger, typically at a certain size, preprocessing a data item becomes faster than reading a data item from the disk. This threshold size depends on many factors such as the speed of the memory, the speed of the disk and the speed of the processors. As a result, this threshold size is hard to predict.

If preprocessing a data item is faster than reading it from the disk, it can be repeatedly generated in each *load* in Algorithm 5. Otherwise, if the data items are written to the disk, the best place to do this in Algorithm 5 is within the loop starting at line 2. Right after preprocessing a data item to determine its size, the data items can be written to the disk before being discarded from the memory.

Since we assume that the memory requirement of a *load* process does not exceed the final preprocessed size of a data item, the choice of reading the preprocessed data item from memory or re-calculating it for each *load* does effect our algorithms. The reason is that Algorithm 4 and 5 allocate total memory required to store the preprocessed data item at the beginning of a *load*. Therefore, these two scenarios have not been addressed separately.

3.5 Experimental Validation

This section implements the proposed algorithm on existing DIAC applications in bioinformatics and experimentally validates the performance of the algorithm. We will introduce benchmark examples first followed by experimental settings and experimental design. Then the experiments and results are presented for all aspects, which will be discussed in experimental design section.

3.5.1 Benchmark Examples

The experiments are conducted on two applications discussed in Section 2.1.3 that have been developed for CV method based calculations [CVTree, 2011, Yu et al., 2010a]. The compared data items in these applications are genomic sequences and the preprocessed data items are called *composition vectors*.

For the experiments, we will use two different datasets that are found in the literature [Yu et al., 2005, 2010a]:

- **Data Set 1**—109 prokaryotes and eukaryotes which are used in [Yu et al., 2005]. These genomic sequences are relatively long. As a result, the sizes of the corresponding *composition vectors* are relatively large and range from 2.4MB to 482.8MB (averaging 214.5MB) when in memory.
- **Data Set 2**—124 large dsDNA viruses used in [Yu et al., 2010a]. These genomic sequences are relatively short. As a result, the sizes of the corresponding *composition vectors* are small and range from 3.0MB to 110.7MB (averaging 17.03MB) when in memory.

Genomic sequences are usually represented using a sequence of single-letter codes. The alphabet for these sequences consist of 4 (for DNA/RNA) or 20 (for Protein)

letter-codes [Tao, 2012]. The datasets that we selected for the experiments in this paper have an alphabet of 20 letter-codes and are stored in FASTA formatted files.

In the following, we apply our memory management algorithms to the two CV method algorithms developed by Yu et al. [2010a] and Qi et al. [2004], which have been described in Section 2.1.3.

3.5.2 Experimental Settings

The following settings are used for the experiments:

Platform:	Linux (Ubuntu 10.04)
Processor:	Core™2 Duo (E8400)
CPU Single Core Speed:	3.00 GHz
Number of Cores:	2
Cache:	6MB
FSB Speed:	1333 MHz
RAM:	4GB
Hyper Threading:	Disabled
HDD Average Read Speed:	62.2 MB/s
K value of k-string:	6
Programming Languages Used:	C/C++

3.5.3 Experimental Design

The following aspects are validated in the experiments:

- the performance of our proposed memory management algorithm versus the virtual memory management by the operating system;

- the performance of the CV method [Yu et al., 2010a] algorithms with and without our memory management optimizations;
- the performance of our memory management approach compared to generic paging algorithms (such as LRU);
- the influence of sorting data items by their size on the performance of the algorithm; and
- the scalability of the proposed solution.

We have improved the original program of Yu et al. [2010a] and have rewritten most of the sections before applying our memory management algorithm. We call the program *Yu et al. refined by us* in which we keep its memory management algorithm unchanged. Then, we apply our memory management algorithm to this program to improve its performance by allowing it to utilize the available physical memory. We call this program *Our Algo. 5 with Yu et al.* For the CVTree application, we have obtained the code from the authors and applied our memory management algorithm to their program to experiment with the performance.

3.5.4 Virtual Memory

The best performance of the program can be expected when all genomic sequences are loaded into the physical memory and are kept in memory until all comparisons are completed as in Algorithm 1. In the process of loading data items, if the program exceeds the available physical memory capacity, the operating system allocates part of the virtual memory to the program. The virtual memory is typically slower than physical memory. Therefore, our memory management algorithm is developed to manage the memory usage within the available physical memory, and the following experiment is designed to prove that it is reasonable to refrain from using virtual memory.

Experiment: A data set of 24 large genomic sequences, which produces *composition*

vectors totalling 7.7 GB after loading into the memory, is used (Physical Memory: 4 GB). In Case I, all genomic sequences were loaded at once, forcing the operating system to manage the memory usage over the physical memory by using virtual memory. In Case II, the loading process of genomic sequences is managed by our proposed algorithm to prevent memory from entering into the virtual memory (see Section 3.5.10 for the techniques to prevent virtual memory).

Results: As seen in Table 3.1, managing memory using our algorithm is extremely fast than utilizing virtual memory (in this case 107 times faster than using virtual memory).

Table 3.1: Comparison between virtual memory managed by the operating system and program itself

	Method	Execution time (sec.)
Case I	Using virtual memory	36,346
Case II	Applying the proposed algorithm	339

3.5.5 Performance of the Algorithm

To validate the performance of our memory management algorithm, we apply it to Yu et al. [2010a] and CVTree [2011] and conduct experiments with different data sets.

Experiment: The versions of the application of Yu et al. [2010a], which are described in Section 3.5.1 are executed with *Data Set 1* and *Data Set 2*. For each run, the execution time is recorded for comparison of performance.

Results: Table 3.2 shows the execution times for different data sets and versions of the application with the physical memory limit imposed. As shown in the results, our algorithm has achieved a dramatic speed-up over the original program (6.5 times faster for *Data Set 1* and 130.5 times faster for *Data Set 2*). Compared to the *Yu et al. refined by us* program, by applying our memory management algorithm, we have

also achieved significant speed-up (2.2 times faster for *Data Set 1* and 31.9 faster for *Data Set 2*).

Table 3.2: Analysing performance of Algorithm 5 with the program by Yu et al. [2010a]

Data Set	Program	Memory limit (GB)	Execution time (sec.)
Set 1	*Yu et al. original*	–	27,968
	Yu et al. refined by us	–	9,434
	Our Algo. 5 with Yu et al.	1.0	5,672
		2.0	4,264
Set 2	*Yu et al. original*	–	29,764
	Yu et al. refined by us	–	7,272
	Our Algo. 5 with Yu et al.	1.0	228
		2.0	220

Experiment: The modified CVTree application with our memory management algorithm and the original CVTree application acquired from the website are executed with *Data Set 1* and *Data Set 2*. For each run, the execution time is recorded for comparison of performance.

Results: Table 3.3 shows the results of the experiment. For the CVTree program we specified the memory limit of 1 GB with *Data Set 1*, and it used a peak memory of 1.35 GB at runtime. After applying our memory algorithm to their program, it always stayed within the specified memory limit. Then, we tested the CVTree application with our algorithm, applying a 1.35 GB memory limit which was the actual peak memory used by the original CVTree program. From the results, it is evident that our algorithm has significantly reduced the execution time of the program, while staying within the specified memory limit (7.2% faster with the 1.35 GB memory limit specified in our program for *Data Set 1* and 17.8% faster for *Data Set 2*).

Table 3.3: Comparing Algorithm 5 with CVTree program's algorithm

Program	Data Set	Imposed Memory Limit (GB)	Execution Time (sec.)	Peak Memory Used (GB)
CVTree original	Set 1	1.00	957	**1.35**
This work	Set 1	1.00	972	0.98
	Set 1	**1.35**	893	1.31
CVTree original	Set 2	0.5	12.6	0.52
This work	Set 2	0.5	10.7	0.45

3.5.6 Comparison with Generic I/O Optimization Algorithms

Experiment: Least Recently Used (LRU) I/O optimization algorithm is applied to *Yu et al. refined by us* application. This version is experimented with *Data Set 2*. Since the performance of LRU algorithm depends on the memory access pattern, we have created two versions of access patterns. In the first version, the access pattern (Access Pattern 1) is similar to Algorithm 1. In the second version (Access Pattern 2), the correlation matrix is divided into equal sized square shaped blocks and comparisons in each block are completed one block at a time. The experiment is also conducted with different block sizes.

Results: Table 3.4 shows the results of the experiment. Compared with the application using LRU with Access Pattern 1, our algorithm achieves 1.6 times faster performance. When compared to the application using LRU with Access Pattern 2, our algorithm still achieves better performance (though the performance improvement is small), when the block size is 45. However, the problem with using LRU with Access Pattern 2 is the difficulty of predicting the optimum block size. When the size is inadequate such as 5 and 55, the performance of LRU becomes very poor in comparison with our algorithm. Adapting a generic algorithm such as LRU in this specific problem does not always guarantee better performance; whereas our algorithm is aware of the context of the problem and always gives good performance.

Table 3.4: Comparing Algorithm 5 with LRU I/O optimization algorithm - *Data Set 2* with 1 GB memory limit

Program	Length of a Square	Execution Time (sec.)
LRU with Pattern in Algorithm 1	-	377.4
LRU with Partitioned Matrix	5	355.7
	15	235.9
	25	232.8
	35	231.8
	45	229.5
	55	304.0
This work	-	228.1

3.5.7 Effect of Sorting of Composition Vectors

Sorting the genomic sequences by the size of their data items affects the execution time. To experiment with the effect of sorting, we have tested Yu et al. program with our memory management algorithm in different sorting orders.

Experiment: Different versions of the memory management algorithm (i.e. amended Algorithm 5) are written with sorting order ascending, descending and without re-ordering. These algorithms are applied to Yu et al.'s program and each version experimented with *Data Set 1* and *Data Set 2*.

Results: Table 3.5 shows the effect of different sorting orders on the computing performance. As shown in the results, when the preprocessed data items are large, sorting in descending order improves the performance. Sorting brings forward bigger data items that take longer time to load. As a result, these data items are loaded fewer times than when they are at the end of the list. Also, the space reserved for the upcoming data item in set B becomes smaller and smaller when proceeding forward in the list. So, the memory available for the bigger set (set A) increases and more data items can be held in the memory.

Table 3.5: Effect of ordering genomic sequences in Algorithm 5

Data Set	Memory (GB)	Sorting method	Execution time (S)
Set 1	1.4	Ascending	9,679
		Unsorted	7,715
		Descending	5,807
Set 2	0.5	Ascending	231
		Unsorted	233
		Descending	231

However, when the data items are smaller, there is no significant variation in the time to load genomic sequences into memory, regardless of their size. So, the sorting order does not make much difference in this situation. It is worth mentioning that the time taken to sort the data items is negligible compared to the execution time of the program.

3.5.8 Performance with Different Memory Sizes

The proposed algorithm is expected to utilize all available physical memory to make the computation faster. So, the execution time should decrease when the size of the available physical memory grows; thus, the solution scales well on different platforms. To validate this, we have conducted the following experiment.

Experiment: Execution times for the same data set (*Data Set 1*) in different available physical memory sizes are recorded.

Results: Figure 3.5 shows how the execution times changes versus the available memory when our algorithm is applied to a program (*Data Set 1* is used with *Our Algo. 5 with Yu et al.* application in this case). The results show that, when the available memory increases, the execution time decreases significantly.

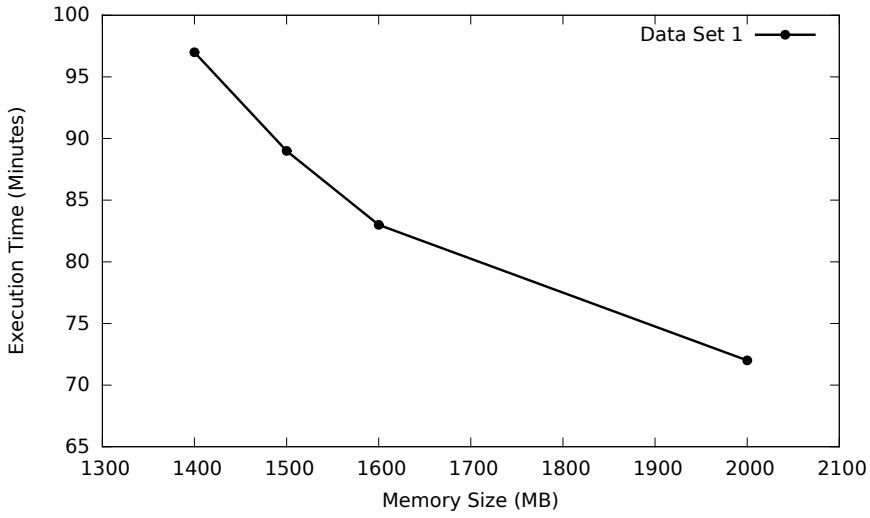


Figure 3.5: Behaviour of the execution time of Algorithm 5 versus the available memory capacity.

3.5.9 Data Structure Used for Composition Vectors

This section briefly discusses the data structure that we used to store the *composition vectors* in the modified Yu et al.'s program (the version "Yu et al. refined by us"). The data structure is depicted in Figure 3.6.

A *composition vector* consists of index/value pairs. In a typical *composition vector*, most of the values are zeros. To avoid wasting memory for storing index/value pairs with the value of zero, we have used a sparse data structure with two arrays. One array stores the indexes of non-zero values (*Array 1*) and the other array stores the non-zero value corresponding to each index (*Array 2*). *Array 1* is sorted in ascending order. Since the algorithm developed by Yu et al. [2010a] accesses the data item indexes sequentially starting from zero, our data structure can quickly determine indexes with zero values using *Array 1*. The non-zero values can also be retrieved quickly with the help of *Array 2*.

When writing a data item to the disk, the two arrays of the data structure are written to a binary file with the size of the arrays (both arrays are similar in size). This allows us to read and write the files faster by mapping the arrays directly to the file.

The CVTree [CVTree, 2011] application uses the standard C++ vector implementation (i.e. std::vector) to store the data items. We did not modify the data structure used in their application even after applying our memory management algorithm.

Array 1
(non-zero indexes)

0	3
1	50
2	345
3	512

Array 2
(non-zero values)

0	1.32
1	2.31
2	1.25
3	1.15

Composition Vector
(excluded indexes are zeros)

3	1.32
50	2.31
345	1.25
512	1.15

Figure 3.6: The data structure developed and used in the applications for CV method, which is based on the solutions from Wang [2009].

3.5.10 Avoiding Virtual Memory

Following methods are used to prevent memory from entering into virtual memory:

1. When *root* permission is available in the computers, the *mlock()* method found in *sys/mman.h* system library is used to prevent virtual memory. When *mlock()* is called on a memory range (typically on a data array), the data is prevented from being *paged* to the swap area (i.e. written into virtual memory). In addition, virtual memory can be completely disabled by using *swapoff -a* command in Linux.
2. When the *root* permission is not available in the computers, previous experiments are used to determine the memory usage threshold on which the virtual

memory comes into play. Keeping system's memory usage below this point avoids virtual memory.

Usually when our applications keep the overall system memory usage 90% or less of the total system memory, the particular Linux version used for experiments did not allocate the virtual memory.

3.5.11 Summary of the Experimental Results

The results in this section so far confirms that the proposed algorithm in this chapter makes the computation of memory-constrained DIAC significantly faster than the existing algorithms. They also demonstrate that the algorithm is a scalable solution with efficient memory management.

In addition, we have analysed how the sorting order of data items before the calculation affects the speed of calculations. It is seen from the experiments that, when the data items are larger in size, ordering them in descending order makes the calculation significantly faster in the experimented applications.

3.6 Summary of the Chapter

This chapter has proposed a novel memory management (paging) algorithm which is both efficient and scalable for DIAC calculation in limited memory. The optimum parameters for the algorithm have also been determined theoretically. The proposed algorithm has been experimentally verified to have better speed compared with the existing memory management algorithms. It has made the computation of existing DIAC applications up to 31.9 times faster starting from 7%. A close-fitting theoretical

lower bound for the number of *loads* required for a DIAC in limited memory has also been derived.

The algorithms proposed in this chapter do not address the challenges of memory management when the tasks are executed in parallel. This will be the main topic of the next chapter.

Chapter 4

Preparation to Solve the DIAC Parallelization Problem

This chapter lays the foundation for presenting a solution for DIAC parallelization problem in Chapter 5 and 6. It first models our DIAC parallelization problem and then extends the model to show that the existing general parallelization techniques are ineffective in solving the problem. This chapter also derives theoretical results useful for designing parallelization algorithms for DIAC.

Over the past few decades, many scheduling techniques for parallel computing have been published. Some of them are potentially applicable to solve our scheduling problem. Five of such existing techniques are evaluated in this chapter based on their ability to solve our DIAC parallelization problem. In the process, the DIAC parallelization problem is formalized and a new model for the problem is proposed. This model is then extended to adapt the existing technique with potential to solve our DIAC problem. The model can also be used in the future researches to solve the problem. Finally, this chapter establishes a theoretical upper bound for the maximum parallel gain of a DIAC under memory constraints. This upper bound is useful to guide design decisions

in the process of developing parallelization algorithms for DIAC and to understand the experimental results of the proposed algorithms.

This chapter provides valuable insights into solving the DIAC parallelization problem and also motivates to propose a new parallel algorithm in Chapter 5. The following incremental contributions are claimed from this chapter:

- The parallelization problem of memory-constrained DIAC is modelled based on several representations.
- Using our models of DIAC, existing parallelization techniques which have potential to solve our DIAC problem are extended and evaluated to solve the DIAC problem.
- A theoretical upper bound for the maximum speed gain from parallel execution of memory-constrained DIAC is derived for shared-memory systems.

4.1 Modelling the DIAC Parallelization Problem

This section evaluates existing techniques which are potentially capable of solving the DIAC problem. To use existing techniques for solving the DIAC problem under memory constraints, two steps are required:

- modelling the DIAC parallelization to be compatible with the target technique and
- adapting/extending the existing technique to solve the DIAC problem under memory limitations.

Therefore, in the following subsections, DIAC is modelled specifically to match each targeted technique and is followed by the adaptation process of the technique. Then the feasibility of using the technique and its effectiveness are evaluated. The evaluations

will show that the methods discussed in this section do not lead to effective solutions for the DIAC parallelization problem.

4.1.1 Modelling as a Scheduling Problem

This section evaluates the feasibility of using solutions proposed for traditional resource constrained scheduling problems for DIAC parallelization. To begin with, it models the parallelization problem of DIAC under memory limitation as a resource constrained scheduling problem. This new model will be used in the rest of the thesis to describe the DIAC problem. This model includes every aspect of solving our DIAC problem in shared-memory systems. After presenting the model, it will be used to identify the difference between traditional resource constrained scheduling problem and our DIAC problem.

In Section 2.1, we discussed the characteristics of the DIAC problem which is addressed in this thesis. In brief, before each comparison, the data items needs to be in memory. The process of bringing the data items to memory (called a *load*) is assumed to be a significantly time-consuming process relative to the pair-wise comparisons. Since memory is limited to hold all data items in memory, some items needs to be swapped in and out from the memory to make room for other data items. We call the operation of deleting an item from memory an unload.

The parallelization of DIAC can be modelled as a resource constrained scheduling problem since the memory, processors and disks as resources limit the concurrent execution of the tasks. In the model, the data items are denoted by G_i where $0 \leq i < N$. Memory and processors are the two main constraints in solving the DIAC parallelization. We have already explained how memory limitation effect the *load* operation. The memory as a resource is required by the *load* operations and also after the *load* operation to keep the data item in memory. The processors are required by each operation

such as a comparison or *load*. Therefore processors as a resource is required for every operation in a DIAC.

Before describing the model, it is worthwhile to briefly discuss the effect of the limitations of the disks in the *load* operation. Some *load* operations use disks intensively. For instance, sometimes in the CV method [Yu et al., 2010a], the pre-calculated data items are read from the disk and the operation requires intensive I/O from the disks. Such disk intensive *load* operations are subjected to the limitations of the disk storage systems.

Some disks can only efficiently handle sequential reads while some can efficiently perform a finite number of parallel reads from the disk. Therefore, the capability of parallel reads of a disk can be interpreted as a resource. For example, if a disk is capable of two parallel reads, then it can be interpreted as two disk resources. When the *load* operations only lightly use disks and the overhead is negligible, the resource of the disks can be considered to be unlimited or very high.

In the model, the dependencies between the operations (*load*, compare, unload) are not expressed explicitly. The dependencies are expressed based on the status of the data items. These dependencies may imply indirect dependencies between the operations.

The Basic Scheduling Model

Each data item can have three statuses: *in memory and in use*, *in memory and not in use* and *not in memory*. If the data item is in either *in memory and in use* or *in memory and not in use* status, it indicates that the item has completed its *load* operation and resides in memory. If the data item is in *not in memory* status, it has never been in memory or has been removed from memory. If it has *in memory and in use* status, the item is being used by one or more comparisons.

There are three types of resources required for a DIAC:

1. R_P represents a processor.
2. R_M represents a byte of memory. $R_M(G_i)$ represents the amount of memory (i.e. R_M) required to *load* G_i and keep it in memory.
3. R_D represents the capability of the disk to do parallel loads. If *load* operations are disk intensive, then R_D is the number of channels that are present in a disk for parallel reads. Otherwise, R_D is considered to be unlimited.

In a DIAC there are three operations (i.e. tasks). Each operation changes the status of a data item and requires certain resources to complete:

1. **Operation** L_i: Represents *load* of G_i. The resources $R_M(G_i)$ of R_M, one of R_P and one of R_D are required. Non-zero t_l time is required to complete the operation. $R_M(G_i)$ of R_M is held even after completing the operation but the other resources are released. Once the operation is completed, G_i goes into *in memory and not in use* status.
2. **Operation** $C_{i,j}$: Represents the comparison between the data items G_i and G_j. One of R_P is required. Non-zero t_c time is required to complete the operation. G_i and G_j go into *in memory and in use* status for the period of the operation or if they are already in *in memory and in use* status they remain in the same status. Upon completion of the comparison operation, G_i goes into *in memory and not in use* status, if and only if G_i does not belong to any active comparison. The same applies to G_j as well.
3. **Operation** U_i: Represents unloading G_i (G_i is deleted from memory). One of R_P is required. The time required to complete the operation is negligible. When the operation is completed, $R_M(G_i)$ of R_M is released and G_i goes into *not in memory* status.

The operation L_i does not release the allocated resource of R_M after completing the

operation (resources R_P and R_D are released). The memory resource (R_M) held after the operation L_i is released by the operation U_i.

The dependencies between the operations are presented based on the status of the data items. To start each following operations, the conditions in the right side must be satisfied.

1. $L_i \Rightarrow G_i$ must be in *not in memory* status.
2. $C_{i,j} \Rightarrow$ Both G_i and G_j must be in either *in memory and in use* or *in memory and not in use* status.
3. $U_i \Rightarrow G_i$ must be in *in memory not in use* status.

The objective is to schedule all comparison operations (i.e. $C_{i,j} \forall i, j \in \{x : 0 \leq x < N\}$ and $i < j$) within a minimum timespan not violating both resource constraints and dependencies. The operations are allowed to be completed simultaneously.

Discussion on the Model

In this model, loading the input data to memory and deleting is split into two operations as L_i and U_i due to the input data sharing between the comparisons. In-between those operations the resource $R_M(G_i)$ is held and the all available comparisons related to G_i (i.e. $C_{i,j} \forall j \in \{x : G_x$ in *in memory in use* OR in *in memory not in use* statuses$\}$) can be completed. Due to the unload (U_i) operation's ability to delete a loaded data item from memory, there can be infinite number of schedules. To avoid this problem it is important but not necessary to complete at least one comparison in-between a *load* (L_i) and unload (U_i) operations. A few sample sequence of operations are listed in Table 4.1. The validity of the sequence according to the model is shown next to each sequence. These samples help better understand the model.

Table 4.1: Sample sequences of operations with their validity according to the model

Sequence	Validity	Status After the Sequence
$L_k \rightarrow L_k$	Invalid	
$L_k \rightarrow U_k \rightarrow U_k$	Invalid	
U_k	Invalid	
$\underbrace{L_k \rightarrow U_k}_{R_M(G_k) \text{ is held}} \rightarrow \underbrace{L_k \rightarrow U_k}_{R_M(G_k) \text{ is held}}$	Valid	G_k is in *not in memory* status
$\underbrace{L_k \rightarrow U_k}_{R_M(G_k) \text{ is held}} \rightarrow \underbrace{L_k}_{R_M(G_k) \text{ is held}}$	Valid	G_k is in *in memory not in use* status
$L_p \rightarrow L_q \rightarrow C_{p,q} \rightarrow U_q$	Valid	G_p is in *in memory and not in use* status and G_q is in *not in memory* status

Using Traditional Resource Constrained Scheduling Techniques

As seen from the model, our scheduling problem deviates from traditional resource constrained scheduling problems. The reason for this deviation is the input data sharing between tasks through the shared memory. If the data sharing is ignored, both operations L_i, L_j will be a part of the operation $C_{i,j}$, and $C_{i,j}$ will require the resources $R_M(G_i) + R_M(G_j)$ of R_M (memory) and 1 of R_P (processor). Most importantly, the acquired amount of R_M will be released as soon as $C_{i,j}$ is completed in this situation. Therefore, when data sharing is ignored, the scheduling problem is similar to a traditional resource constrained scheduling problem where the resource required for a task is held throughout the task and released upon completion of the task.

4.1.2 Self-Adjusting Dynamic Scheduling (SADS)

This section evaluates the effectiveness of SADS technique to solve our DIAC problem. SADS family algorithms are powerful for solving various parallelization problems. As we described earlier in Section 2.5, SADS is based on branch and bound search for a solution. Unlike the branch and bound algorithm, SADS algorithms do

not find a complete schedule by searching all possible solutions. Instead, whenever a processor become idle, the branch and bound search stops and a best solution developed until then is picked from the already developed incomplete but feasible schedules. Then the processors are scheduled with the newly found solutions. The branch and bound search continues from that point until next time a processor becomes idle.

When adopting SADS approach to solve our problem, there are four key problems to address:

1. How to model a DIAC schedule in a search tree structure.
2. What are the decision points where new branches of the search tree start from?
3. What different operations (branches) possibly start at each decision point?
4. How to select the best node from the partial schedules.

The Basic SADS Model

To solve the DIAC problem using the SADS technique, the scheduling of DIAC tasks must be modelled based on a search tree. Our model is an extension of the model proposed by Hamidzadeh et al. [2000] for their SADS algorithm. In their algorithm, each node of the search tree completes only one task-processor mapping since they target heterogeneous processors. Instead, our model merges several task-processor mappings into one node as we are dealing with homogeneous processors. This method helps to identify duplicate nodes before spending time to process or extend them. Merging is possible since the execution time of a task does not depend on the processor which executes the task. In our model, a new node is added only at a decision point which is described in the next section. A portion of an example presentation of a DIAC using our tree model is depicted in Figure 4.1.

The Decision Points: A decision point is a node in the search tree. One or more

processors are assigned new tasks at a decision point. Once scheduling starts, an initial task mapping for the processors must be performed. Since memory is initially empty, the first mapping will have only *load* operations. Thereafter, next decision point can be at any point because the processors can wait without starting a task. This way infinite number of decision points can be created by applying various wait times for each processor. To prevent this, the next decision point is assumed to be at the next system status change. The status of the system changes only when a *load*, comparison or an unload is completed by any of the processors. Since, new decisions become available only after the system status changes, this decision is reasonable.

It is useful to understand the reasons why completion of each of *load*, unload and comparison (tasks) triggers a system status change. An unload changes the status of the system by removing an item from memory, although it has a negligible execution time. Completion of a *load* brings more comparisons to memory and changes the system status. A completed comparison could potentially free up locked items which were locked to memory while performing the comparison, therefore causes a system status change. In addition, all tasks completions free a processor and which changes the potential of the system to complete work.

Decisions Taken at a Node: At each decision point there can be four different operations that can be started by a processor; *load*, compare, unload and wait.

The following process is used to create children of a node at each decision point. The feasibility of each task (e.g. sufficient memory and disk resources for a load; both data items are in memory for a comparison) is evaluated and feasible tasks are added to a list. Then, all free threads at the decision point are identified. Thereafter, non-repetitive combinations of length which equals to the number of free threads are created from the list of feasible tasks. For example, if there are 3 free threads and 5 feasible tasks, non-duplicate combinations of size 3 are created using the 5 tasks. In the following we will further discuss each of the four actions.

Load: The load operation is a special operation as it is bounded by the available memory and disk resources. The feasibility of a *load* always depends on the availability of memory and disk resources. For example, let us assume that the *load* operations of G_1, G_2 and G_3 are feasible and the data items are dissimilar in size. Memory is sufficient to load G_1 and G_2 together but if G_3 is loaded first, there is no space for either G_1 or G_2 to be loaded. Given that two processors are free at the decision point, only the combinations listed below are feasible (please note that both processors are identical). L_x represents *load* operation of G_x, and W represents making the thread wait until the next decision point:

Processor 1	Processor 2
L_1	L_2
L_3	W

If all possible combinations are considered:

Processor 1	Processor 2	
L_1	L_2	
L_1	L_3	Infeasible and produces either $L_1\ W$ or $L_3\ W$
L_2	L_3	Infeasible and produces either $L_2\ W$ or $L_3\ W$

As seen in the example above, it is important to avoid duplicate combinations which could result from memory constraints. If a node has only one task-processor mapping, duplicate schedules will be created when the schedule is extended further, as seen in the last two combinations above.

Comparison: A comparison is only feasible if both data items required are in memory. Once a comparison is started, both items required for a comparison will be locked to the memory until the comparison is completed. A locked item cannot be unloaded.

Unload: Unload is considered as a special event. At each decision point (i.e. node), unloads are performed to form all possible combinations from data items in the mem-

ory by unloading one or more items for each combination. For example, assume that in the current node, G_1, G_2 and G_3 are in memory and no item is locked. The combinations are as follows.

G_1 G_2 G_3
G_1 G_2
G_1 G_3
G_2 G_3
G_1
G_2
G_3

For each of these combinations, a duplicate of the current node is made and unload operations to produce each of the combinations will be applied on the duplicates (as seen in Node (b) to (e) and (d) to (h) in Figure 4.1). Then each child is extended from that point onwards. When extending the schedule, the new nodes which are duplicates of the existing nodes in the tree are eliminated. Two nodes that have all following similarities are called duplicates in our model:

- same uncompleted comparisons
- same data items in memory
- same on-going tasks each of which have spent same time to the similar task in the other node.

In Figure 4.1 node (e) is a duplicate of (a), and (h) is a duplicate of (c). The time of the node can be different in two duplicate nodes.

Wait: the wait operation has two different applications. First one is when the maximum possible operations in a combination are less than the number of free threads, the rest of the threads must wait. The other type of waiting is used to preserve the branches that could lead to better schedules. For example if there are two possible loads G_1 and

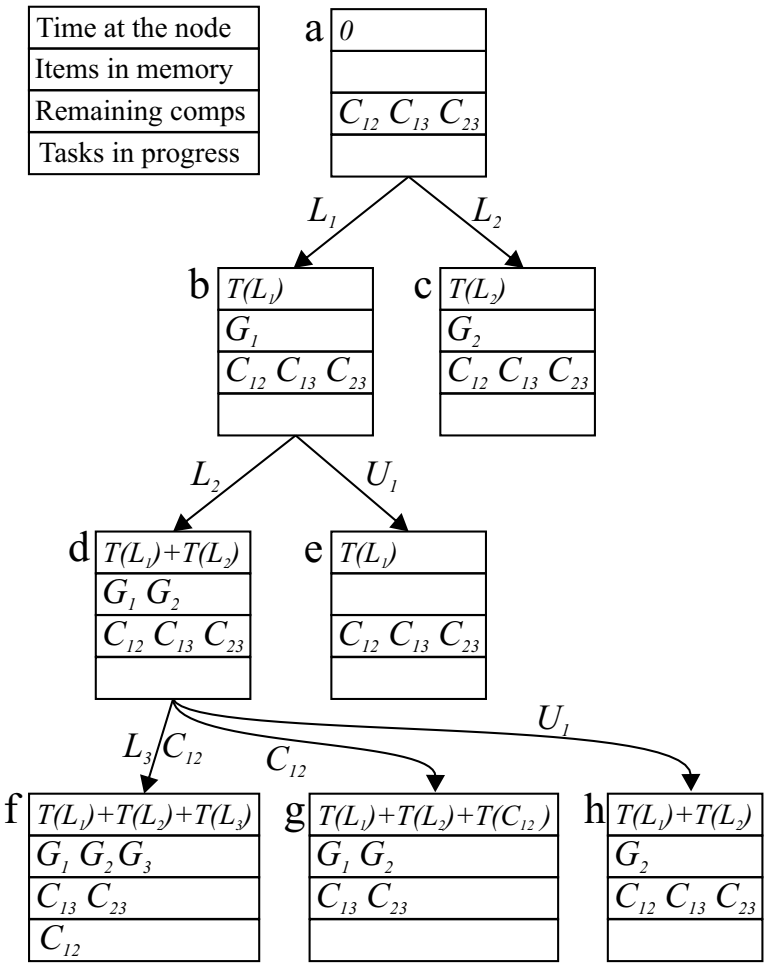


Figure 4.1: A sample search tree generated for a DIAC with settings $N = 5, P = 2, L_P = 1$ and $M = 3$. Please note that only a few branches at each level are shown and only the **comparisons within first three data items** are shown in the nodes for simplicity. L_x represents the load of G_x; C_{xy} represents the comparison between G_x and G_y; and $T(X)$ represents the time required for the task X; $T(L_3) < T(C_{12})$.

G_2, and there are two free threads, loading G_1 and G_2 in the two threads seems to be the only possible combination. However, there are three more combinations. Those are:

Processor 1	Processor 2
L_1	W
L_2	W
W	W

In those combinations, the threads are waiting until the next decision point, although there are sufficient combinations available to accommodate all free threads. Also, if there are other busy threads, all free threads can wait until the next state changes. We call this kind of waiting as *extreme waiting* (e.g. node (g) in Figure 4.1). *Extreme waiting* may lead to better schedules. In particular, it could be efficient to avoid some unnecessary *loads* or comparisons, in order to make a schedule faster. For example, starting the last comparison with a data item in memory blocks the unload operation of the data item until the comparison is completed. Since only one comparison has a relationship to the data item, it could be efficient to unload it and then *load* another data item that could potentially make more threads busy.

Solving the Problem Using the Model and SADS

The feasibility of using SADS algorithm to solve the DIAC problem (using the model that we developed) is significantly restricted by the long execution time in scheduling. In our experiments (based on Data Set 1 and Data Set 2 described in Section 3.5.1), the SADS method exceeded the runtime of the sequential program only on scheduling. The number of combinations at each node is a typical factor for the fast execution of the SADS algorithm. In our DIAC problem, memory constraints and sharing data among tasks makes the number of children of a node considerably high compared to scheduling independent tasks in an unconstrained situation. The following mathematical development demonstrates that the number of nodes at each level is extremely large.

The following equation can be used to calculate the number of children of a node, N_{ch}, in the worst case scenario assuming that the data items are uniformly sized (with *extreme waiting*).

$$N_{ch} = \underbrace{\sum_{r=1}^{m} \left[\frac{m!}{(m-r)!r!} \right]}_{\text{Combinations created from unloads}} + \underbrace{\sum_{r=1}^{P_f} \left[\frac{t!}{(t-r)!(r!)} \right]}_{\text{Combinations created from available } \textit{loads} \text{ and comparisons}} \tag{4.1}$$

where P_f is the number of free processors at the node, m is the number of data items in memory and t is the number of all feasible tasks which is:

$$t = \underbrace{L_{feasible}}_{\text{Number of feasible loads}} + \underbrace{\frac{m(m-1)}{2}}_{\text{Number of comparisons}} \tag{4.2}$$

$L_{feasible}$ is defined as:

$$L_{feasible} \begin{cases} = N - m; & m < M \\ = 0; & m = M \end{cases} \tag{4.3}$$

The following example gives a rough idea of the number of children of a node in a typical scenario. If $N = 100, M = 5, m = 4$ and $P_f = 4$ the number of children in the node will be 4.426×10^6. This is a large number of child nodes and in many real scenarios where the dataset is large, SADS algorithm fails to produce a sufficiently long incomplete schedule before the processors become idle. This is a major drawback of employing SADS to solve our DIAC problem. The memory constraints and data sharing among tasks significantly lowers the performance of SADS by creating a huge number of possibilities at each node. Because of the huge number of children created in each level, SADS algorithm becomes an inefficient method to deal with real datasets. In the experiments, most of the time the scheduling algorithm could not complete a single level of the search tree before a processor become idle.

4.1.3 Directed Acyclic Graphs (DAG)

In recent years, DAGs have become popular to handle scheduling problem with dependencies [Abdelkader and Omara, 2012, Amalarethinam and Mary, 2011, Meng et al., 2013]. Since the comparisons have a dependency on *loads* in our problem, this section investigates the feasibility of DAG to our scheduling problem.

The Basic DAG Model

The first step of solving a problem using DAG is to represent the problem in a DAG. Figure 4.2 shows a graphical representation of our problem in a DAG, ignoring the memory constraints. The nodes (i.e. circles) represent the tasks and the directional edges (i.e. arrows) represent the dependencies. A task pointed by an arrow depends on the task from which the arrow starts. L_i represents the *load* of G_i and $C_{x,y}$ represents the comparison between G_x and G_y. As seen in the figure, the *Data Intensive All-to-all Comparison (DIAC)* can be represented by a DAG.

Under the memory constraints, the DAG shown in Figure 4.2 is still valid. However, a constraint has to be applied over the DAG, which is the sum of the data items loaded to memory cannot exceed the memory capacity. Assuming that all data items are similar in size and the maximum of M data items can be held in memory, more than M *loads* cannot be completed without unloading a data item in between the *loads*. As a result, the solutions based only on the DAG are not valid under this constraint.

Solving the DAG

None of the existing work based on the DAGs [Abdelkader and Omara, 2012, Amalarethinam and Mary, 2011, Meng et al., 2013] have considered a DAG with extra

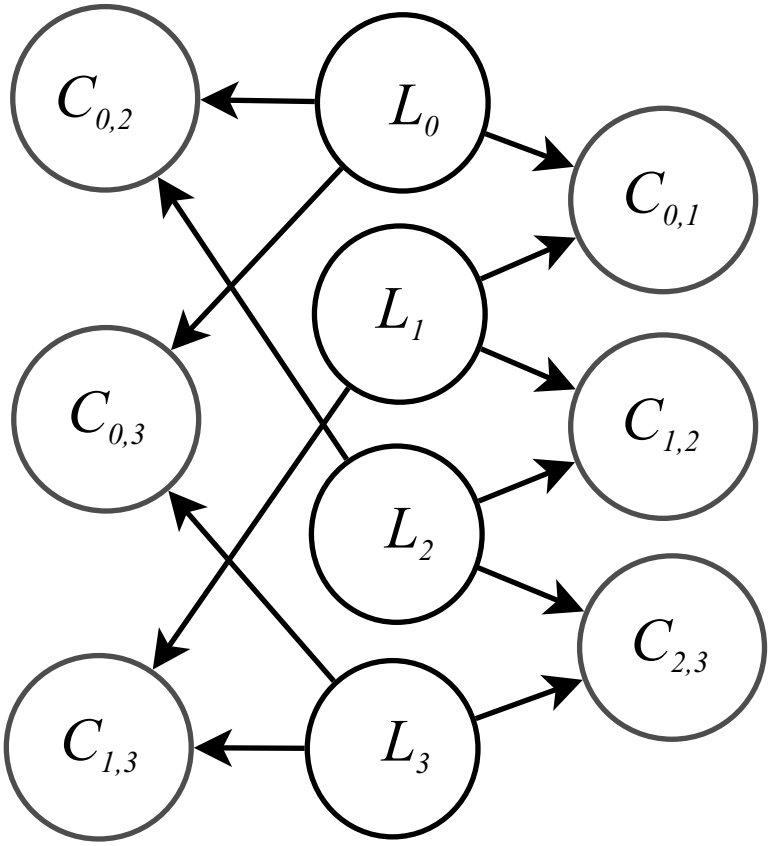


Figure 4.2: The DAG represents our DIAC parallelization problem assuming unlimited memory available.

constraints for parallelization. Therefore, existing methods based on the DAG cannot be applied to our problem.

4.1.4 Heuristic Functions Based Scheduling

A solutions based only on heuristic functions to make scheduling decisions on DIAC parallelization is evaluated in this section. Heuristic functions have been used very frequently to solve parallelization problems in the past [Berman et al., 1999, Casanova et al., 2000, Giersch et al., 2004, Ramamritham et al., 1990, Xiangbin and Shiliang,

2003]. As we described in Section 2.5, a heuristic function calculates a representative value based on the current state of the system, to help the scheduler to decide the task-processor affinity or next action. Heuristic functions are usually developed to represent a combination of one or more ideas which are likely to affect the objective.

Since there is no existing heuristic function based method which is capable of handling memory constraints in the DIAC problem, a new model is developed in this section. This model combines multiple strategies aimed to make DIAC parallelization efficient and build a single decision making approach. This approach was the basis for building our proposed pattern based algorithm presented in Chapter 5 (Algorithm 7). Some of the key decisions taken in the task completion pattern for Algorithm 7 are inspired by the results of these heuristic functions. A comparison of the decisions made by heuristic functions and Algorithm 7 developed in Chapter 5, under a similar status in the system will be conducted at the later part of this section. However, the model only based on the result of the heuristic function is live-lock prone and computationally intensive in runtime. Therefore, this section aims only at building a decision making model based on heuristic functions to help identify a data loading/unloading pattern to solve DIAC parallelization problem.

The Basic Heuristic Function Model

Since our objective is a combination of managing memory, data sharing and increasing speed, many of the conventional heuristics functions such as 'longest job first' cannot be used efficiently. Therefore, a new model based on five decisions is developed first. Each decision to start a task (load, comparison and unload) must be sure to meet the constraints in the model presented in Section 4.1.1. The following are the decisions to be made using heuristic functions:

- Should a load be started?

- Should a comparison be started?
- Which data item should be loaded next?
- Which comparison should be started next?
- Should one or more data items be unloaded and what are they?

A decision-making procedure for DIAC parallelization based on the these decisions is depicted in Figure 4.3.

Solving the Problem using Heuristics

The heuristics used for making the decisions are summarized in five tables as follows. As seen in the heuristic functions, they are designed to meet the constraints in the model presented in Section 4.1.1.

- Table 4.2—Should a load be started?
- Table 4.3—Should a comparison be started?
- Table 4.4—Which data item should be loaded next?
- Table 4.5—Which comparison should be started next?
- Table 4.6—Should one or more data items be unloaded and what are they?

In each table, there are several different heuristics presented for each decision. The combined final value of the heuristics for a decision will be a weighted average of each of the sub heuristics. Each heuristic function returns a value between 0 and 1.

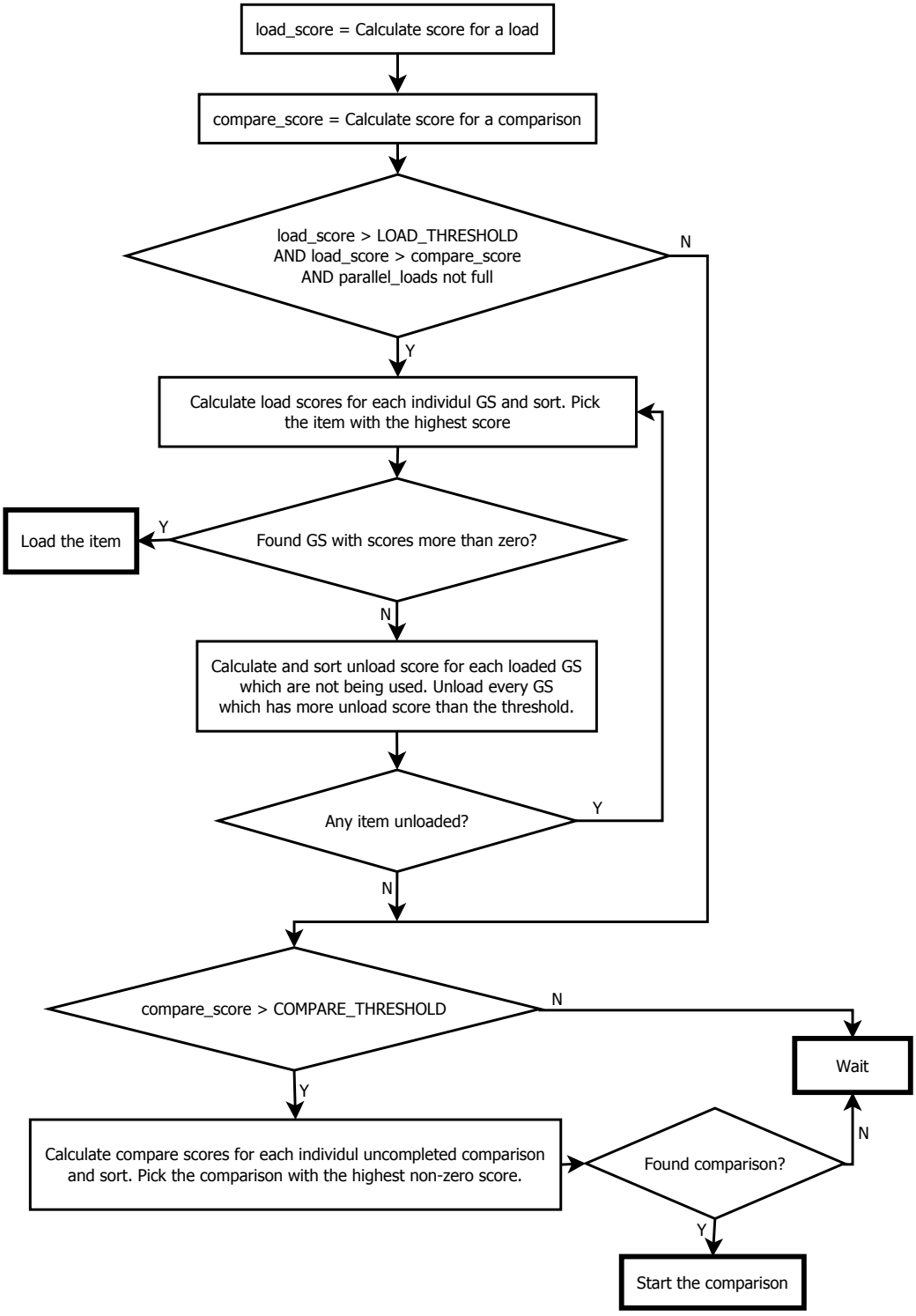


Figure 4.3: Procedure for making decisions based on the heuristic functions

Table 4.2: Heuristic functions used for the decision of starting a *load*. The heuristics to decide which data item must be loaded is discussed in Table 4.4

Heuristic Function	**Description**
Sufficient space to *load* the data item	To *load* a data item there should be sufficient memory available. If this condition is not met, the heuristic function for the decision returns 0.
More free memory: $$h_2 = \frac{\text{Free Memory}}{\text{Total Memory}}$$	To make full use of available memory, it has to be full most of the time throughout the calculation. Therefore, more free memory encourages *loads*.
Small number of available comparisons: Let the number of currently available comparisons be C_{avail} and maximum possible available comparisons which may reside in the memory be C_{max_avail}. $$h_3 = 1 - \frac{C_{avail}}{C_{max\_avail}}$$	Small number of available comparisons indicates the need for more possible *loads* to increase the availability of comparisons for threads.

Table 4.3: Heuristic functions used for the decision of starting a comparison

Heuristic Function	**Description**
Higher number of summation of available comparisons and number of busy threads: Let the number of currently available comparisons be C_{avail}; maximum possible available comparisons which may reside in the memory be C_{max_avail}; the number of currently busy threads be P_{busy} and the total number of threads be P. $$h_1 = \frac{C_{avail}}{C_{max\_avail}} + \frac{P_{busy}}{P}$$	More busy cores suggests that more comparisons are being completed. Therefore, if the summation of available comparisons and busy threads is high, it suggests that more comparisons are pending to be completed. Therefore, comparisons are encouraged in this situation.
One or more comparisons are available between memory locked items: $$h_2 = \begin{cases} 1; & \text{If available} \\ 0; & \text{Otherwise} \end{cases}$$	When comparisons are available between the data items which are already locked to memory (comparisons are ongoing with them), it is a good opportunity to start another comparison without conceding memory. Therefore, if comparisons between memory locked data items are available, those are encouraged to be completed.
More number of data items are locked to memory: $$h_3 = \frac{\text{Locked to memory}}{\text{Total loaded}}$$	When more data items are locked to memory, comparisons must be encouraged to take advantage of already locked data items if possible.

Table 4.4: Heuristic functions used for the decision of what to load

Heuristic Function	Description
Memory is sufficient to load.	Memory must be sufficient to load a data item. Therefore, if this condition is not met, the final value of the heuristic for the data item will be zero.
More comparisons brought in: Let the number of comparisons brought in by loading the data item be C_{in} and the maximum possible comparisons could be brought by loading a data item be C_{in_max}. $$h_1 = \frac{C_{in}}{C_{in\_max}}$$	If a data item can bring more comparisons to memory than others, it has the advantage of making more threads busy with less memory consumption.
More relationships to uncompleted comparisons: Let the number of relationships the data item has with the uncompleted comparisons be R_{uncomp} and the total number of data items be N. $$h_2 = \frac{R_{uncomp}}{N}$$	Loading a data item that has more relationships to uncompleted comparisons increases the chances of its future contribution to bringing in more comparisons.

Table 4.5: Heuristic functions used for the decision of what to compare

Heuristic Function	**Description**
Both data items of the comparison are in memory	If this condition is unsatisfied the final value of the heuristic function for this decision is zero.
Number of items locked to the memory: Let the number of data items locked to memory be N_{locked}. $$h_1 = \frac{N_{locked}}{2}$$	When more data item of a comparison is already locked to memory (i.e. comparisons are ongoing with the same data items), memory will be shared among more comparisons by starting the comparison. This increases the efficient utilization of memory.
More connections through both data items of the comparison to the already available comparisons. Let the number of connections from both data items to the already available comparisons be R_{avail} and the number of data items currently loaded to memory be m. If $m <= 1$, $h_2 = 0$. $$h_2 = \frac{R_{avail}}{2m - 2}$$	Starting a comparison which has more shared data items with other available comparisons increases the chances of completing the comparison parallel to the other connected comparisons and releasing the data items quickly.
Small chance of re-introduction (data items do not relate to many uncompleted comparisons). Let the number of connections to each uncompleted comparison through both data items be R_{ucon} and the number of data items to be compared be N. $$h_3 = 1 - \frac{R_{ucon}}{2N - 2}$$	The comparisons are less likely to be brought into memory again if it has only few shared data items with uncompleted comparisons. On the other hand, since the comparison does not share data items with many uncompleted comparisons, completing it later may require *loads* dedicated for the comparison.

Table 4.6: Heuristic functions used for the decision of unloading data items

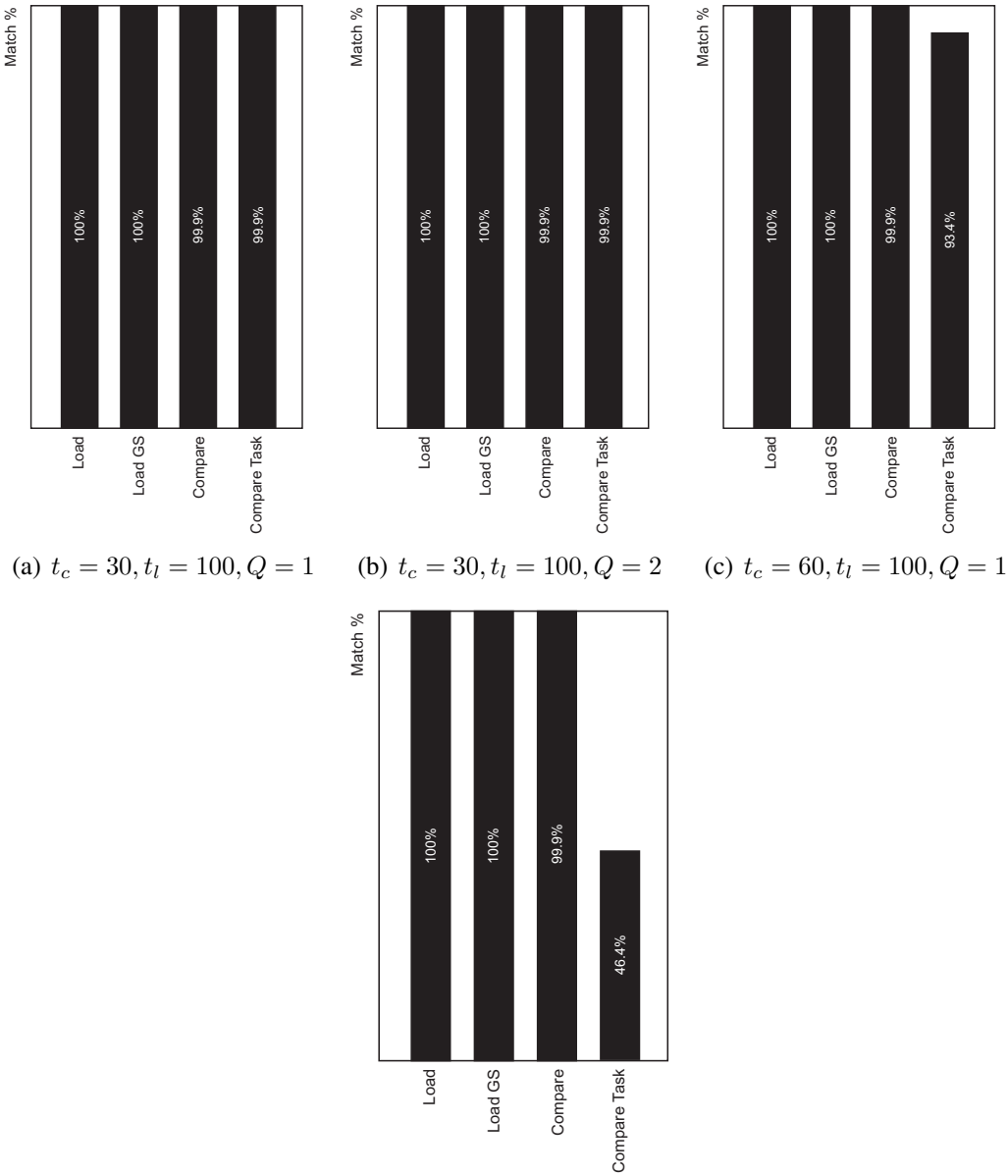
Heuristic Function	Description
Not locked to memory. If the data item is locked to memory final value of the heuristic function for this decision will be 0.	If a data item is being used by a comparison, it cannot be unloaded.
Low number of relationships with uncompleted already available comparisons. Let the number of related comparisons to the data item be $C_{related}$ and the total available comparisons be C_{avail}. $$h_1 = 1 - \frac{C_{related}}{C_{avail}}$$	If a data item is unloaded while more uncompleted comparisons in memory, there is a high chance of wasting the *load*.
Low number of relationships with uncompleted but half available comparisons (one data item in memory). Let the number of relationships of the data item to the uncompleted and half available comparisons be R_{half} and the total half available comparisons be C_{half}. $$h_2 = 1 - \frac{R_{half}}{C_{half}}$$	If the data item has relationships to uncompleted comparisons which has one item already in memory, there is a good chance of more comparison becoming available with the data item soon.
Low number of relationships with uncompleted comparisons. Let the number of related uncompleted comparisons to the data item be R_{ucomp} and the total number of uncompleted comparisons be R_{total_ucomp}. $$h_3 = 1 - \frac{R_{ucomp}}{R_{total\_ucomp}}$$	If many comparisons related to the data item is still uncompleted, there is a good chance of a comparison becoming available soon with the data item.

The output of each of the heuristic functions is a good indication of the next proper decision to make, as it will be seen from the algorithm (presented in Chapter 5) based on the decisions. The idea of developing these heuristics was inspired by the work of Giersch et al. [2004]. However, our approach is different from their approach since they only use one criterion in every heuristic function, while we combine a few criteria for each decision. On the other hand, to solve our DIAC problem we have to make more decisions rather than just deciding the processor task affinity and the order of tasks.

However, an algorithm purely based on the heuristic scores tends to be live-lock prone although the scores are good indications of the next proper decision to make. This is because the decisions taken from the heuristics are sometimes short-sighted and lead to more greedy solutions and end up in never ending cycle of task loops. On the other hand, most of the listed heuristics have higher computational complexity compared to our pattern based scheduling algorithm (i.e. Algorithm 7) which will be presented in Chapter 5. Because of this, an algorithm based only on these heuristics spends more time on scheduling.

Therefore, we didn't use the this heuristics based model as a scheduling algorithm in runtime. Instead, the scores calculated from the heuristic functions were used to help designing the loading pattern used in Algorithm 7. This is because the decisions based on the heuristics are, most of the time, good indications of the proper decision to make. Figure 4.4 shows the percentage of the matches of the heuristic functions based decisions taken by the model (depicted in Figure 4.3), to the decisions taken by Algorithm 7 at a similar system status.

As seen in the figure, only two scenarios Figure 4.4 (c) and (d) show a deviation of the heuristic functions based decisions from the decisions made by Algorithm 7. This deviation is mainly in picking up the correct comparison when the queue size is greater than one. In this situation, the heuristic functions tends to complete the comparisons



(a) $t_c = 30, t_l = 100, Q = 1$ (b) $t_c = 30, t_l = 100, Q = 2$ (c) $t_c = 60, t_l = 100, Q = 1$

(d) $t_c = 60, t_l = 100, Q = 2$

Figure 4.4: Comparison of decision making based on heuristic function to Algorithm 7 in different settings. Setting are $N = 200$ and $M = 20$.

loaded last as all comparisons in set B have similar scores. When several comparisons have similar scores to the highest, picking up the first comparisons in the available comparisons list is specific to the implementation. The decisions made by the *Unload* heuristic functions do not match with the decisions of Algorithm 7. The reason is that the most of decisions to unload data items made by the heuristic are short sighted and cause redundant loads. Therefore, we ignored this heuristic functions scores while developing Algorithm 7. Instead, we were able to adapt the algorithms developed in Chapter 3 successfully to handle unloads in Algorithm 7.

4.1.5 Using Local Search with Traditional Memory Management Techniques

This section develops an approach to solve the DIAC using a local search technique. The aim is to develop an algorithm regardless of the time taken for scheduling, to produce a schedule as a reference for designing a new algorithm. Very briefly, the strategy is to repeatedly modify a schedule over a long period of time to approach a refined solution closer to the optimum. SADS [Hamidzadeh et al., 2000] and heuristic function [Giersch et al., 2004] approaches failed to produce a conclusive solution even after running for a long time. Therefore, this section proposes a method by combining traditional memory management algorithms such as First In First Out (FIFO) with simulated annealing [Shroff et al., 1996] to search for a solution in a long run. The new model used in this section is an extension to the model proposed in Section 4.1.4.

First In First Out (FIFO), Least Frequently Used (LFU) and Least Recently Used (LRU) are some of the popular memory management techniques used for paging data in memory. When we evaluated performance of these algorithms specific to solving our memory management problem (in Chapter 3), we found that the performance of the algorithms depend on the data access pattern of the DIAC application. For in-

stance, we developed a data access pattern for DIAC in Chapter 3 by tiling the matrix into squares and carrying out comparisons in each square one after another. In this algorithm, when the comparisons in the matrix were tiled to the correct size, the LRU algorithm performed nearly as fast as our memory management algorithm.

To achieve a better solution, our aim is to progressively improve a schedule to produce data access patterns better suited for the memory management algorithm. We use a technique based on simulated annealing [Shroff et al., 1996] to progressively improve the memory access pattern by using random modifications.

In simulated annealing, an initial candidate solution is built and successive random modifications are made to the solution. A fitness value which represents the suitability of a solution is used to determine the acceptance of the solutions. If a modified solution is accepted, it is retained and the current solution is discarded. A variable called temperature is used to control the acceptance of the modified solutions. The temperature parameter is gradually decreased (cooled) at each time a new solution is created. Solutions with better fitness value will be accepted always. If a solution has a lesser fitness value than the current solution, it may be accepted with a probability directly related to the current temperature. When the temperature cools down, the probability of accepting less fit solutions decreases. In the long run, the simulated annealing algorithm is able to reach towards an optimum solution [Shroff et al., 1996].

The Basic Model

With regard to our DIAC problem, a solution is a complete schedule which completes all comparisons without violating the resource and task constraints in the model presented in Section 4.1.1. Typically in simulated annealing the current solution is randomly modified to create the next solution. However, in our case, a schedule cannot be modified always by delaying tasks or interchanging tasks, because of the tasks that

depend on the previous tasks. Such modification violates the feasibility of the schedule. For instance, if a *load* is delayed as a result of a modification, all comparisons related to the *load* must be delayed. Then, the tasks following the comparisons must be delayed too, which results in a completely different schedule.

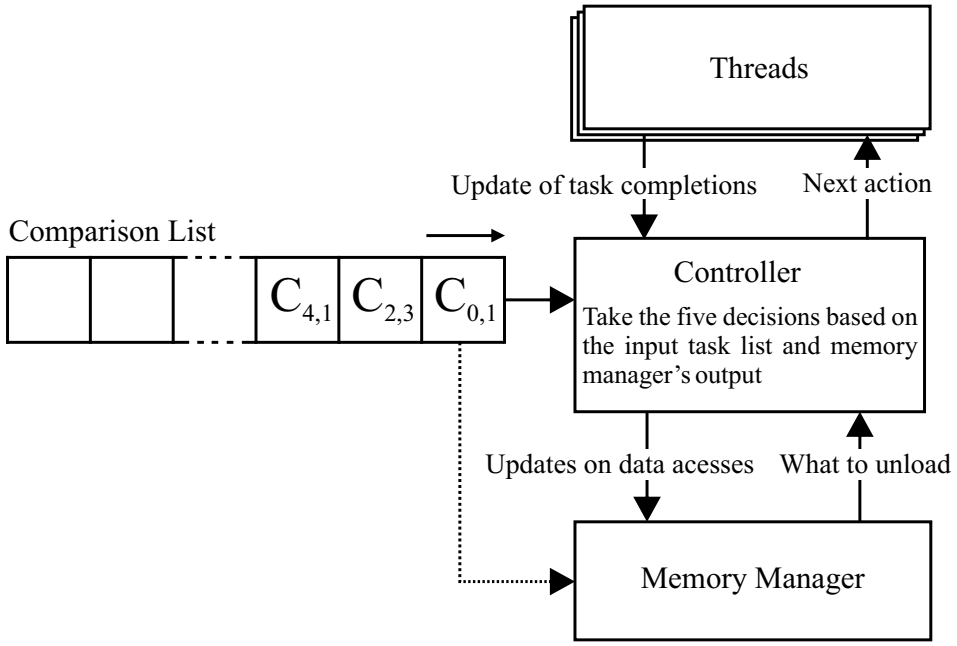


Figure 4.5: Model for producing schedules based on the order of a supplied sequence of comparisons.

To overcome this issue, we use the order of the completion of comparison as the base for developing a schedule. The model is depicted in Figure 4.5. The controller takes the five decisions of the model described in Section 4.1.4. It is not allowed to violate the given sequence of comparisons while building a schedule. To progressively modify the memory access pattern, for each repeated run, the order of the comparisons are randomly modified by swapping two positions in the sequence. The schedule produced by using the new comparison sequence is slightly different from the current schedule.

If the memory is insufficient to *load* a data item which is required to start the next comparison in the supplied list, the controller requests an ordered list of data items to

be unloaded (best to the least best) from the memory manager. The memory manager uses an algorithm like LRU. It has access to the order of comparisons in the given sequence and receives updates on data item access. Based on the list of data items to be unloaded and returned by the memory manager, the controller takes a decision as described in the next section.

Solving the Problem Using Traditional Memory Management Techniques

We have experimented with two different variations of the memory management algorithms. The first one is the LRU algorithm which performs well in the experiments conducted in Chapter 3. The second algorithm is called the Future Aware (FA) algorithm, which we designed specifically for this situation based on a widely accepted "informal principle of optimality" for page replacement. The principle is that the page to be replaced is that which has the longest expected time until next reference [Aho et al., 1971]. As we are aware of the order of completing comparisons, the next soonest referral for each data item can be determined easily. Therefore, before unloading a data item, the FA algorithm can decide the optimum data item to be unloaded.

Then we developed two variations of both the LRU and FA algorithms because the data item which is most favoured by the memory management algorithm to unload can be sometimes in use.

- The first variation waits until the most favoured data item completes its comparisons and then unloads it. The threads have to wait until the most favoured data item becomes free to unload, and then *load* the data items required by the next comparison in the sequence.
- The other variation considers unloading only the data items which are not locked to memory. It picks the most favoured data item from the non-locked items as suggested by the memory management algorithm, and unloads it immediately.

Finally, we extended the later variation (unloading immediately) of LRU and FA algorithms to make it flexible when following the given sequence of comparisons. In this extended algorithm, whenever it is not possible to start the next comparison in the given sequence of comparisons, threads are allowed to start other available comparisons. However, as soon as the next comparison is available to be started, the next free thread must start it. Which data item is loaded next is still determined by the comparisons in the given sequence. The data item to be loaded next is always one of the data items required for the next comparison in the supplied sequence. Therefore, changing the order of the input comparisons sequence still affects the schedules created.

Table 4.7 shows the total runtime of the best schedule produced by each variation after two days and the runtime of the schedule produced by Algorithm 8 presented in Chapter 6. The LRU and FA variations strictly follow the given sequences of comparisons. The *Fast LRU* and *Fast FA* variations are flexible in following the given sequence of comparisons as described in the previous paragraph. *Wait* and *No Wait* variations of above algorithms show weather the algorithms wait and unload the best data item to be unloaded or unload the best data item which is not busy immediately.

The results show that after two days of runtime, *Fast FA* algorithm found a solution very close to Algorithm 8 for Data Set 2. It also beats Algorithm 8 by 3.6% for Data Set 1. In the Data Set 1, Algorithm 8 cannot make all threads busy due to the parallel loads restriction. In this kind of situation, there can be a slightly better solutions for arranging the comparisons than how Algorithm 8 does. More importantly the data items have non-uniform sizes and can result many different combinations that can be slightly better than Algorithm 8. However, it should be noted that these SA algorithms took 2 days to find a better result while Algorithm 8 spent less than 1.5 and 3 seconds for Data Set 2 and Data Set 1 respectively in the simulator. Section 3.5.1 describes the data sets and only one parallel load is allowed in these tests.

Table 4.7: Best runtime of schedules produced by the different variations of generic memory management algorithms combined with simulated annealing after **two days** of running (The number of maximum efficient parallel loads, $L_P = 1$).

Data Set	Variation	Pseudo memory limit (MB)	Best schedule's runtime (s)
Data Set 1	LRU (No wait)	5120	17186.8
	LRU (Wait)	5120	17319.1
	FA (No wait)	5120	16167.1
	FA (Wait)	5120	16219.4
	Fast LRU (No wait)	5120	4488.2
	Fast FA (No wait)	5120	2710.0
	Algorithm 8 in Chapter 6	5120	2812.3
Data Set 2	LRU (No wait)	500	31.39
	LRU (Wait)	500	31.61
	FA (No wait)	500	30.65
	FA (Wait)	500	62.06
	Fast LRU (No wait)	500	15.73
	Fast FA (No wait)	500	15.71
	Algorithm 8 in Chapter 6	500	14.53

4.2 Upper bound for the Parallel Performance Gain

This section focuses on deriving a theoretical upper bound of the gain in speed by using parallel execution when a system is only capable of sequential *loads*. Due to limitation of the disks, computers which are only capable of sequential *loads* (i.e. inefficient in concurrently reading data) are commonly found. To derive the upper-bound, we assume that a computer with unlimited number of processors exists. Although, this assumption is not realistic for many applications, the result derived from the theoretical analysis provides good estimations of the theoretical upper bounds for system performance, and are useful to guide system design and implementation.

The results are summarised in two theorems. Theorem 9 derives the theoretical minimum time required to complete an *all-to-all comparison* with parallel execution. Then Theorem 10 establishes the theoretical upper bound for the maximum gain in speed with parallel execution.

Theorem 9. *Let M denote the maximum number of data items can be held in memory, t_l the time required for a load and N the number of data items in a data set. Lower bound of the time required to complete an all-to-all comparison in a shared-memory computer which is only capable of sequential loads is t_{pmin}, where,*

$$t_{pmin} = t_l \left(\left\lfloor \frac{N(N-1)}{2(M-1)} - \frac{M}{2} + 1 \right\rfloor + M - 1 \right). \tag{4.4}$$

Proof. The proof is based on the Theorems 6, 7 and 8 established in Section 3.3.2.

After parallelization, an algorithm at least requires time to complete minimum required *loads*, even if the time required to complete the comparisons is ignored (if all comparisons are available in memory, they can be completed simultaneously since there are unlimited number of processors). As a result, the *loads* can be carried out sequentially one after another in the best case. Theorem 8 established the minimum number of *loads* required for an DIAC. In the process, it considered a sequence that brings maximum possible number of comparisons at each *load*. A similar sequence brings comparisons at the maximum rate possible at each stage since it brings the maximum possible comparisons within t_l at each stage. The minimum number of *loads* required to complete an *all-to-all comparison*, according to Theorem 8 is:

$$L_{tmin} = \left\lfloor \frac{N(N-1)}{2(M-1)} - \frac{M}{2} + 1 \right\rfloor + M - 1. \tag{4.5}$$

Therefore, lower bound of time required to complete a DIAC with parallel execution t_{pmin} is:

$$t_{pmin} = t_l L_{tmin}. \tag{4.6}$$

Substituting Equation (4.5) into Equation (4.6) gives Equation (4.4). This completes the proof. □

Theorem 10. *Let t_l denote the time required for a load and, t_c the time required for a comparison. An upper bound of the maximum speed gain, S_{max}, of an all-to-all comparison by using parallel execution in a shared-memory computer capable only of sequential loads, compared to the best possible sequential all-to-all comparison algorithm running in a uni-processor system (assuming that no tasks can be executed in parallel) with similar properties is:*

$$S_{max} = 1 + \frac{Nt_c(N-1)}{2t_l\left(\left\lfloor \frac{N(N-1)}{2(M-1)} - \frac{M}{2} + 1 \right\rfloor + M - 1\right)}. \tag{4.7}$$

Proof. The minimum time required to complete an *all-to-all comparison* without parallelism, t_{smin}, is the time spent on minimum required number of *loads* (from Equation (8)) and the time to complete the comparisons. Therefore,

$$t_{smin} = t_l * L_{tmin} + \left(\frac{N(N-1)}{2}\right) t_c \tag{4.8}$$

From Equation (4.4), the best parallel algorithm at least consume t_{pmin} time according to Theorem 9. Therefore,

$$S_{max} = \frac{t_{smin}}{t_{pmin}} \tag{4.9}$$

Substituting Equations (4.8) and (4.4) into Equation (4.9) gives Equation (4.7). This completes the proof. □

Figure 4.6 shows a comparison between speed-up of our proposed parallel algorithm (Algorithm 8 in Chapter 6) and the upper bound of speed-up (from Theorem 10). The speed-ups are calculated relative to the best possible sequential algorithm's time from

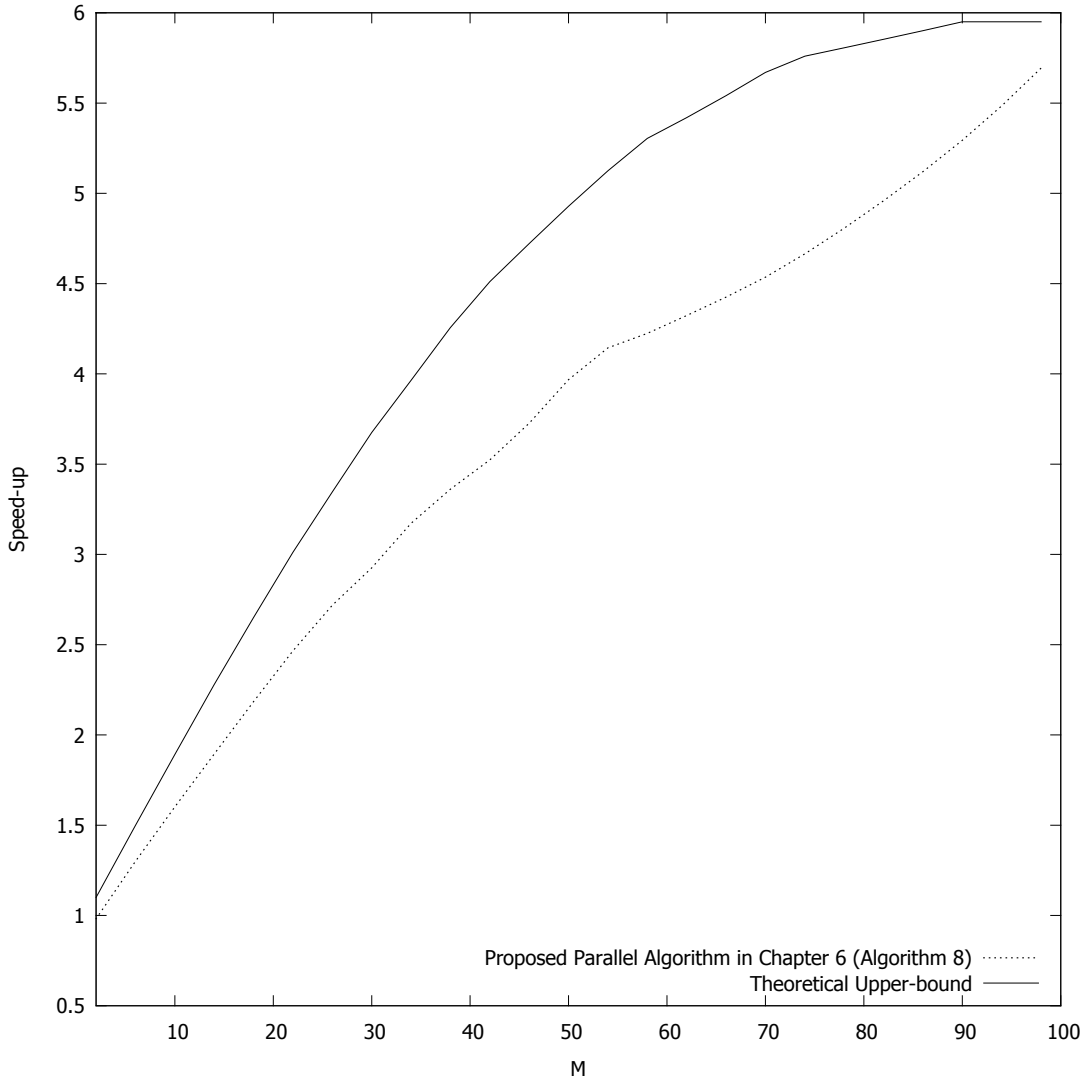


Figure 4.6: The speed-up of Algorithm 8 (presented in Chapter 6) and upper bound of speed-up (from Theorem 10) plotted versus M for $N = 100, t_l = 100$ and $t_c = 10$. The speed-ups are calculated relative to the best possible sequential algorithm's time from Equation (4.8). 100 threads are used in the simulation of Algorithm 8

Equation (4.8). The number of processors are assumed to be unlimited which is the basic assumption of Theorems 9 and 10. As seen from Figure 4.6 of the speed-up, the actual speed-up is bound by the theoretical upper bound. The upper bound for the parallel gain is caused by the limitation of memory and the data throughput.

The theoretical results obtained in this section show that there is an upper bound for the speed gain of a DIAC under limited memory and disk throughput regardless of the availability of processors. The theoretical result will be important for understanding some of the experimental results in Chapter 6 which shows a limited speed-up regardless of the number of available processors.

4.3 Summary of the Chapter

This chapter modelled the DIAC parallelization problem as a resource constrained scheduling problem. Additionally, the model was extended to four new models to be used with existing parallelization techniques to solve the DIAC problem. Some existing parallelization techniques were capable of solving the problem after extending to use our models.

The effectiveness of each existing parallelization technique was evaluated based on the solutions created by adapting the technique. Our DIAC problem deviates from traditional resource constrained scheduling problems as seen after modelling it as a resource constrained scheduling problem. The SADS based solution spends unacceptable time in scheduling a DIAC with a typical sized dataset in limited memory. It is unable to handle large data sets due to the exponentially increasing complexity with the size of a data set. The existing work for DAG based parallelization [Abdelkader and Omara, 2012, Amalarethinam and Mary, 2011, Meng et al., 2013] cannot solve the DIAC parallelization problem presented in a DAG because of the constraints added by memory limitations. The solutions purely based on heuristic functions tends to be live-lock prone and unable to produce a complete schedule. It also spends significant time on scheduling due to the complex heuristic functions. The local search technique developed for DIAC by combining simulated annealing and our Future Aware (FA) memory management algorithm created a successful schedule after two days of runtime.

Although, these techniques are not effective in solving our DIAC problem, the process of adapting existing techniques and modelling DIAC developed useful insights into solving the problem. For instance, the Heuristic Function model produced a valuable reference decision making process to solve the problem and the local search model developed a reference schedule after running for a long period of time. The lack of effectiveness of the existing techniques seen in the chapter motivated us to develop a novel parallelization technique by eliminating the problems in the current techniques. The next chapter will develop this novel algorithm.

This chapter also established a theoretical upper bound for the maximum parallel gain which can be achieved by parallelization of DIAC under memory and disk throughput constraints. The theorem provides insight into the limitations in designing parallelization algorithm for DIAC. The upper bound can also be used to explain the outcomes in experimental results of parallelization algorithms for DIAC.

Chapter 5

Parallelization and Memory Management in Multi-core Systems

As we have seen in Chapter 3, the problem of managing memory to minimize runtime alone (without parallelization) is challenging to solve. Parallelization of DIAC in limited memory is an even more challenging problem to solve due to the added complexity of *load* balancing. Chapter 4 showed that none of the existing parallelization techniques which have potential to solve our DIAC problem are effective enough under memory and data throughput constraints. The parallelization must address both memory management and task parallelization. The underlying memory management has a significant influence on the quality of the scheduling. Therefore, this chapter focuses on developing a combined memory management and parallelization algorithm for DIAC in limited memory. A novel parallelization technique is developed in the process of developing the algorithm. This algorithm further extends and utilizes the algorithms developed in Chapter 3.

The algorithm development is based on several key strategies designed to overcome the gaps identified in Chapter 4. Firstly, each of these strategies are presented paired

with the challenge to overcome. Then, the algorithm is presented as a step-by-step development explaining utilization of underlying strategies. Then, the optimum parameter settings for the algorithm are derived theoretically for the commonly found system parameters.

This chapter contributes a novel parallelization technique which combines static and dynamic scheduling. The static scheduling is guided by a specific pattern developed based on the prior knowledge of the abstracted problem. This technique significantly minimizes the time spent on scheduling for complex parallelization problems such as DIAC where significant prior knowledge of the abstracted problem is available (e.g. data dependency and task information). Implementing this technique, a novel scheduling algorithm for the memory management and parallelization of DIAC in shared-memory multi-core platforms is designed. The algorithm is empirically shown to reach very close to the maximum utilizations of processors for comparisons. In addition, the optimum parameter settings for the proposed algorithm are also theoretically derived for commonly found scenarios.

The experiments and implementation details of the algorithm proposed in this chapter is presented in Chapter 6.

5.1 Development of Scheduling Strategies

We have identified four key aspects for solving the DIAC scheduling problem. Each aspect is addressed in a subsection.

In this section, the term 'redundant *loads*' is used often. The term is defined as follows. Let the minimum number of *loads* required to complete a DIAC under a memory limitation be X. If an algorithm to complete the DIAC performs L *loads* where $L \geq X$,

under the same memory limitation, there are $L - X$ number of 'redundant *loads*' in the algorithm.

5.1.1 Aspect 1: Maximizing Comparisons and Minimizing Loads

In this aspect there are two challenges to overcome. Therefore, two strategies are developed to overcome the challenges.

Challenge 1: Handling resource sharing between the *loads* and comparisons is complex. Each *load* has a cost associated with it because a *load* requires memory and a thread allocated as soon as the operation started. However, due to memory limitations, multiple *loads* of some data items are essential to complete all comparisons. On the other hand, completing more *loads* does not necessarily complete the target workload (comparisons). Therefore, the balance between *loads* and comparisons is important to efficiently handle parallelization of DIAC. To address this aspect of the DIAC problem, the following strategy is introduced.

Strategy to Overcome Challenge 1: The strategy is to balance the rate of bringing comparisons to memory and the rate of completing comparisons. This strategy is utilized as the key for developing further strategies to address other aspects as well.

Finding the best balance between the rates of bringing comparisons and completing comparison is difficult. Let P, t_c and t_l be the number of threads, average time for a comparison and average time for a *load* respectively. In the best case, $t_c(M-1)$ work can be loaded to the memory within t_l time, if we assume no parallel loads are possible. In the best case, by P threads Pt_c work can be finished within t_c time. However, the problem is that these best rates cannot be maintained continuously. The reason is that while n number of *loads* are being done, only $P-n$ threads are available for comparisons and other threads might have to wait until a *load* is completed to start

comparisons. Therefore, the strategy for balancing the rates is that to first decide the rate of *loads* which can just keep all threads busy with comparisons (if possible) and still try to avoid redundant *loads* as much as possible.

When the items are read from a disk that is incapable of parallel reads, the *loading* of items usually becomes the bottle neck. In this case, *loads* have more adverse affect on the total runtime, since the *loads* have to be carried out one after another, and more *loads* directly increases the runtime. Therefore, balancing the rate of *loads* and completing comparisons properly is even more important when parallel *loads* are impossible.

Challenge 2: There are two key limitations in parallelization of DIAC due to the memory constraints:

1. The number of comparisons available in memory.
2. The number of comparisons a *load* can bring into memory.

The two limitations are explained in the following.

If the maximum number of data items which can be held in the memory is denoted by M, then the maximum number of comparisons which can be available in memory is $M(M-1)/2$. This happens only if the comparison between the set of data items loaded are not yet completed. On the other hand, a single *load* can bring the maximum of $M-1$ new comparisons to memory. This occurs only when $M-1$ data items are already in memory and none of the comparisons between the newly loaded data item and the data items already in the memory have been completed.

The two limitations mentioned previously limit the amount of work available in memory at a given time. They also limit the rate of bringing new comparisons to memory. Therefore, these limitations need to be taken into account in the parallelization DIAC.

Strategy 2: The strategy to overcome this problem is intelligently choosing the combination of data items remaining in memory and continuously changing it to bring new comparisons in a flow. Whenever possible, the combination of data items in memory should try to provide sufficient work to the threads while still allocating resources for *loads* to continue the flow of incoming new comparisons.

To match the rate of completing comparisons, the rate of loading data items should be constantly monitored to avoid starvation of threads for comparisons. If the continuous change of combination of data items in memory could be performed with fewer redundant *loads*, there is a good chance of developing a quality schedule.

5.1.2 Aspect 2: Using the Knowledge of DIAC Problem

As we have abstracted the DIAC and memory management problem clearly, there is useful knowledge available about the characteristics of the problem. The knowledge of the tasks and their data dependencies are available in advance to be used for making the scheduling decisions.

Mueen et al. [2010] and our algorithm developed for memory management in Chapter 3 utilized a pre-defined data loading pattern to minimize the scheduling overhead in the runtime. This method can significantly reduce the scheduling overhead compared to a completely dynamic algorithm. For example, consider the SADS dynamic scheduling algorithm [Hamidzadeh et al., 2000, Hamidzadeh and Lilja, 1996] which is based on a branch and bound search. As we saw in Section 4.1.2, such scheduling algorithm suffers from a significant dynamic scheduling overhead compared to the pattern based algorithm proposed in Chapter 3. Therefore, it is important to take advantage of already available knowledge prior to the runtime to reduce dynamic scheduling overhead.

Even though we have complete knowledge of problem abstraction there is information that is not available prior to the runtime. Importantly, we do not have accurate estimations for the completion-time of either *loads* or comparisons. Although the *load* time can be measured prior to the runtime, it may vary due to unexpected random fluctuations of the computing platforms during the runtime.

Strategy: The strategy to address this aspect is to utilize a dynamic scheduler guided by a static schedule. A specific pattern to complete *loads* and comparisons needs to be developed with three aspects in mind. The pattern must:

- be able to dynamically adjust the rates of completing comparisons and loading comparisons;
- be able to withstand on fluctuation of estimated runtimes for the tasks; and
- should encourage continuous execution of comparisons.

The development of the pattern to *load* and complete comparisons utilizes the knowledge available prior to the runtime. The pattern acts as a static schedule guiding the dynamic scheduling.

5.1.3 Aspect 3: Minimizing Synchronization Overhead

In our DIAC problem, the *load* times and comparison time are not uniform. As a result, two comparisons which are started at the same time might finish at different times. This makes centralized scheduling inefficient as some of the threads might have to wait at barriers until other cores finishes the jobs. To minimize this synchronization overhead, following strategy is introduced.

Strategy: The strategy is to distribute decision making into the threads. Each thread is responsible for making decisions of its behaviour based on the guidelines and the state

of the system. The thread can decide to start a *load* or comparison, or wait until the state of the system changes (i.e. wait until a comparison or *load* is completed, or until one or more data items are unloaded from the memory).

Distributed scheduling also helps overlap the scheduling process without dedicating a separate core for scheduling only. In addition, this method promotes thread reuse since the threads do not die until all comparisons are completed. The threads themselves decide the next action each time when it completes a job or ends a wait (a wait ends when a waiting thread is notified by another thread). Thread re-use prevents the overhead of repeated thread creation in the operating system (OS). It also helps to improve overall system performance.

5.1.4 Aspect 4: Time Spent on Scheduling

The computational complexity of a scheduling algorithm is a factor which has a significant effect on overall runtime. Even if a scheduling algorithm can produce quality schedules, if its complexity is high it may make the threads wait for a long period. Therefore, it is important to focus on reducing the complexity of the proposed scheduling algorithm. As already mentioned, the help of the static algorithm can significantly reduce the dynamic scheduling overhead in the runtime. We trimmed Algorithm 5 (see Chapter 3) so that the complexity could be reduced without sacrificing significant performance loss. Unlike in the single-core algorithm, the scheduling algorithm runs very often in our multi-core design, since every thread runs a self decision making process. Therefore, our aim is to keep the computational complexity of the algorithm at a minimum level.

5.2 Algorithm Design

Based on the strategies developed in Section 5.1, this section designs an algorithm. The designing begins with developing a task (*loads*, unloads and comparisons) completion pattern. The final algorithm is then developed based on the pattern. The algorithm design is presented progressively starting from naive parallel algorithms to our final algorithm. Therefore, several intermediate algorithms are presented in the process of developing our final algorithm.

A new constant, L_P is introduced for parallel algorithms, which is mainly calculated based on the performance of the disk system of the host computing platform. L_P represents the number of efficient parallel *load* operations. If *load* operations are disk intensive, L_P directly correlates to the number of R_D introduced in Section 4.1.1 (i.e. the number of disk resources in a system). In this case, L_P is the number of channels which are present in a disk for parallel reads similar to R_D. Otherwise, L_P is considered to be limited only by the available memory.

Determining the value of L_P is based on the ability of a hard disk system to read the disk in parallel. The value of L_P is generally 1 for most of the commonly found hard drive systems. We assume that L_P is an integer based on the number of separate reads a disk system can perform. For example, if the target computing platform has a RAID (Redundant Array of Independent Disks) drive [Feng et al., 2010] with two independent drives, L_P is considered to be two. This is under the assumption that a data read rate of $F \times L_P$ can be achieved by L_P number of threads, if a single thread can achieve a data read rate of F.

Before presenting the first parallel algorithm, it should be noted that the choice of reading pre-written data items from the disk or re-calculating them for each *load*, does not effect the parallel algorithms. As we discussed in Section 3.4, we assume that

a *load* operation does not require more memory than what is required to keep the final preprocessed data item in memory. Similar to memory management algorithms targeted uni-processor platforms, our parallel algorithms also allocate memory for the preprocessed data item before the *load* operation begins. Therefore, the only difference between to two scenarios is how L_P is determined. When data items are re-calculated for each *load*, the value of L_P is only limited by the available memory because the *load* operations are not disk intensive. However, when the data items are read from the disk for each *load*, L_P is determined based on the system's disk performance.

5.2.1 Designing the Pattern

Our initial attempt is to apply parallelism to the algorithms proposed in Chapter 3, which manage the memory efficiently for the DIAC (Algorithm 5). Algorithm 5 forms groups of rows in the correlation matrix, which are called rows in the algorithm as seen in Figure 5.1. Height of a row in the algorithm is $M - 1$ (i.e. the row is formed by combining $M - 1$ rows of the matrix), where M is the maximum number of data items that can be held in the memory. Since the sizes of data items are not uniform, M varies based on the combination loaded to the memory. Therefore, the rows may have different heights.

In Algorithm 5, each row is divided into two sections as set A and set B. Set A is the first $M-1$ data items in the row, and they are not unloaded until all comparisons in the row is completed. The rest of the data items in the row are one-by-one loaded into set B until all comparisons in the row are completed. However, the last data item loaded to set B in a row is kept in memory and will be reused as the next row. This reusing is called 'bringing forward' mechanism.

When we parallelize Algorithm 5 the 'bringing forward' mechanism is ignored. This is done to keep the complexity of the algorithm at a minimum level and to improve

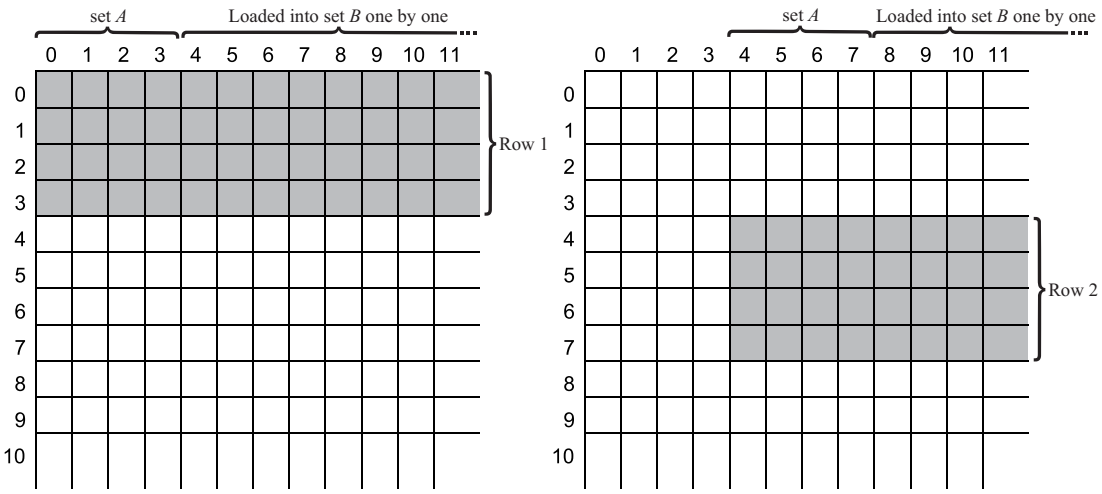


Figure 5.1: How the correlation matrix is divided into rows and how a row is divided into two sets set A and set B

the ease of further extending the algorithm. The cost of extra *loads* caused by ignoring 'bringing forward' mechanism is nullified by the reduced parallel scheduling overhead as we will see later in this section. Since there is no 'bringing forward', all data items loaded for a row are unloaded from memory before a new row begins (i.e. memory is emptied before each row begins). The resulting algorithm is closer to the one proposed by Mueen et al. [2010] (see also Algorithm 3) except that the rows in our case have non-uniform heights.

In Algorithm 5, the comparisons are completed after each *load*. The simplest way to parallelize this algorithm is as follows. While loading set A, one or more threads *load* the data items and the rest of the threads complete the newly introduced comparisons as they are loaded. The number of threads allocated for loading set A is limited by the maximum parallel *loads* allowed (i.e. the maximum parallel reads that can be done efficiently in the system). Once set A is completely loaded, the rest of the data items are loaded to set B one at a time by a thread. Each time a new data item is loaded to set B, all threads start completing the comparisons. While loading a data item to set B, all other threads except the thread loading the item wait until the *load* is complete.

The above described parallel algorithm is depicted in Algorithm 6. In this section, M is considered to be a constant, for the simplicity of presenting and explaining the intermediate algorithms. However, these algorithms can be modified to handle variable M by simply modifying the margins of set A and set B. We initially present intermediate algorithms assuming that M is a constant but our final algorithm will be presented with a variable M.

Algorithm 6 Pseudo-code representation of the simple parallel algorithm based on Algorithm 5; N is number of items; M is maximum number of items fit into memory; L_P is maximum number of efficient parallel loads; P - is number of threads.

```
1:  procedure CompleteTasks(N, M, P, L_P)
2:      r ← 0
3:      while r < N do
4:          A ← {G_x : x ∈ ℤ, r ≤ x < r + M}
5:          [L_P Threads] Load all items in {x : x ∈ ℤ, r ≤ x < r + M}
6:          [P − L_P Threads] Complete comparisons as introduced  // Run parallel to step at line 5
7:          for all {i : i ∈ ℤ, r + M ≤ i < N} do
8:              [Single Thread] Load G_i
9:              [P − 1 Threads] Complete remaining comparisons within set A  // Run parallel to step at line 8
10:             [All Threads] Complete all new comparisons with G_i
11:             [Single Thread] Unload G_i
12:         end for
13:         [All Threads] Complete all remaining comparisons within set A
14:         [Single Thread] Unload all in set A
15:         r ← r + M − 1
16:     end while
17: end procedure
```

To explain the behaviour of Algorithm 6, snapshots of *thread* and *matrix* diagrams are shown in Figure 5.2. Before using these diagrams, the keys to read the diagrams are as follows.

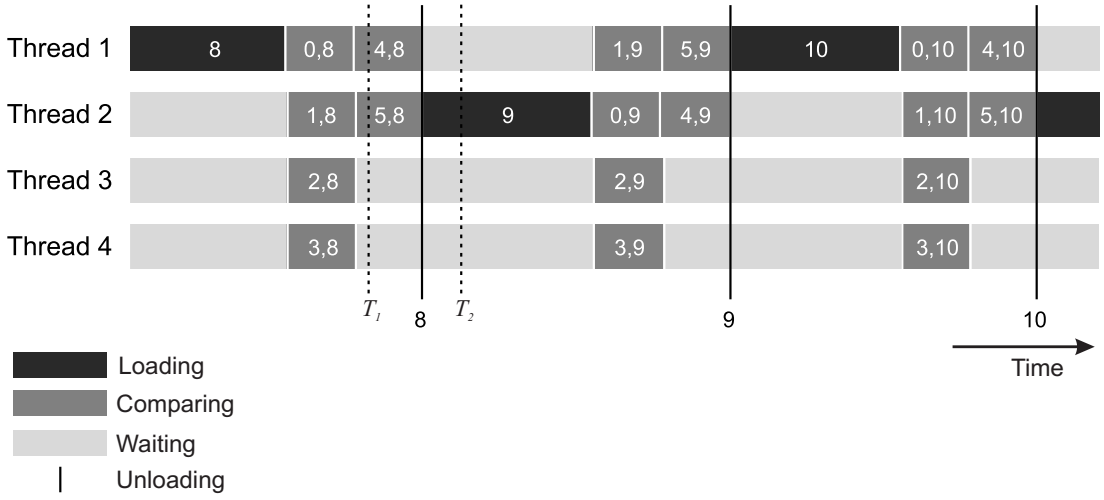
Figure 5.2 (a) shows how each thread behaves over the time in a DIAC. Each row in the graph shows what task (*load*, compare, unload or wait) the thread is performing at a given time. The task can be distinguished by referring to the colour in the legend.

Since, an unload consumes a negligible amount of time, it is shown as a vertical line at the time it occurred. The data item number or numbers involved in each action is shown inside it.

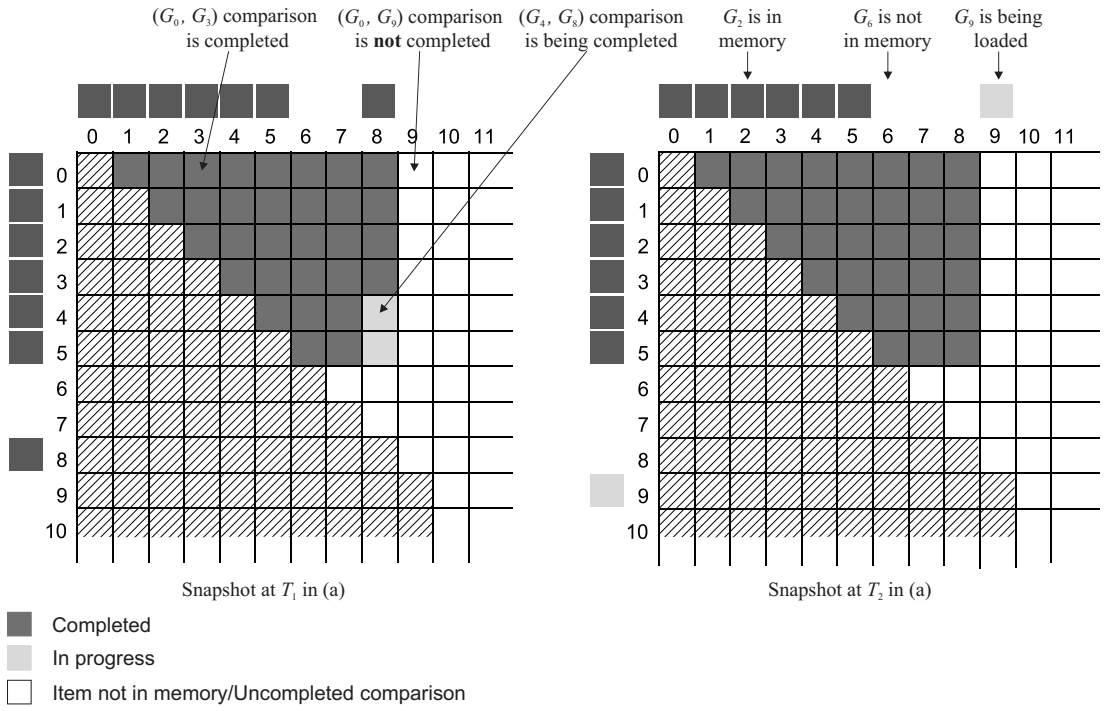
A matrix diagram is used to show the system status at a moment of time. Figure 5.2 (b) shows two snapshots of the system status at T_1 and T_2 marked in Figure 5.2 (a), respectively. A matrix diagram shows a part of the correlation matrix. In the matrix, both columns and rows are numbered from 0 to $N-1$, where N is the number of data items to be compared. Alongside the top and left of the column and row numbers (outside the matrix), the status of each data item is shown in a square. If the square i is filled with dark grey (labelled "Completed" in the legends), the data item G_i is in memory (i.e. the data item is loaded). If the square is white, G_i is not in memory and if it is light grey (labelled "In progress" in the legend) G_i is being loaded. For each data item, the status is shown in both the column and row for faster referrals.

In the matrix diagram, each square within the matrix represents a comparison between the data item numbered as in the row and column, i.e. COMPARE(G_{row}, G_{column}). The correlation matrix is symmetric and the diagonal is zeros. Therefore, the comparisons to be completed take the shape of an upper triangular matrix without the diagonal. The lower triangular matrix and the diagonal in Figure 5.2 (b) are hatched to indicate that the comparisons in that area are not required to be completed. This hatch is not shown in the future matrix diagrams, but it must be noted that only the comparisons in the upper triangular matrix are required to be completed. The matrix diagram only shows the status of the comparisons in the upper half of the matrix for clarity. If a comparison shown by a square is white, then the comparison is not yet started; if it is colour "In progress", the comparison is being completed; and if it is colour "Completed", the comparison has been completed.

Algorithm 6 has the advantage of a lower number of *loads*. As we will be discussing, there are situations where this algorithm is better due to low number of *loads*. In the



(a) Thread Diagram



(b) Matrix Diagram

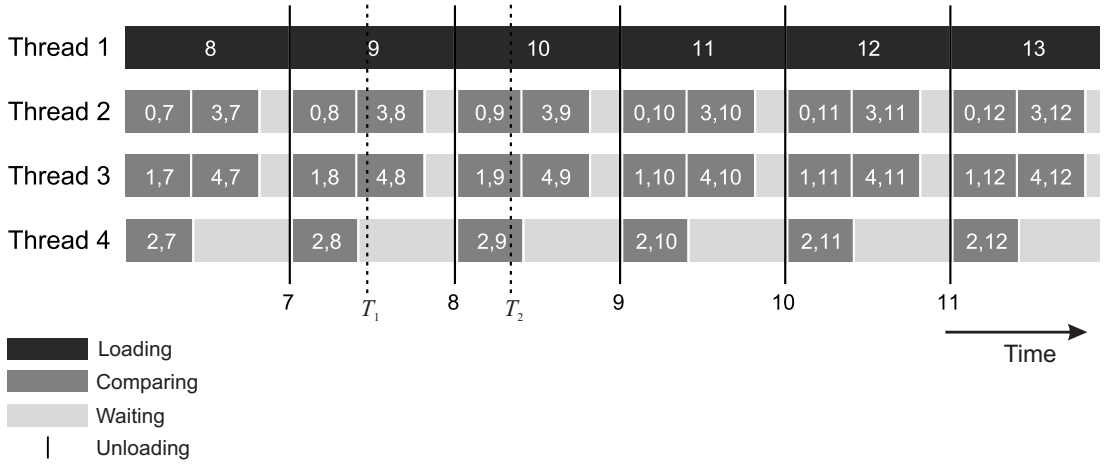
Figure 5.2: The behaviour of Algorithm 6 explained in a matrix diagram and a thread diagram. The settings are $N = 20, M = 7$ and $P = 4$.

current row of the algorithm shown in the Figure 5.2, set A is $\{G_0 \dots G_5\}$ according to Figure 5.2 (b). In Algorithm 6, the threads have to wait until the *loads* are completed after loading set A, as shown in Figure 5.2 (a). This wait time increases even further if the *load* time is considerably longer than the comparison time. For the whole period, while each *load* is performed, only one thread is busy and that thread is also not performing useful work (useful work is comparisons). This prolonged wait time is one of the two disadvantages of this algorithm. The other disadvantage is that it cannot take the advantage of parallel *loads*. After loading set A, memory is just sufficient for a data item only. Therefore, sufficient memory is not left for two or more parallel *loads*.

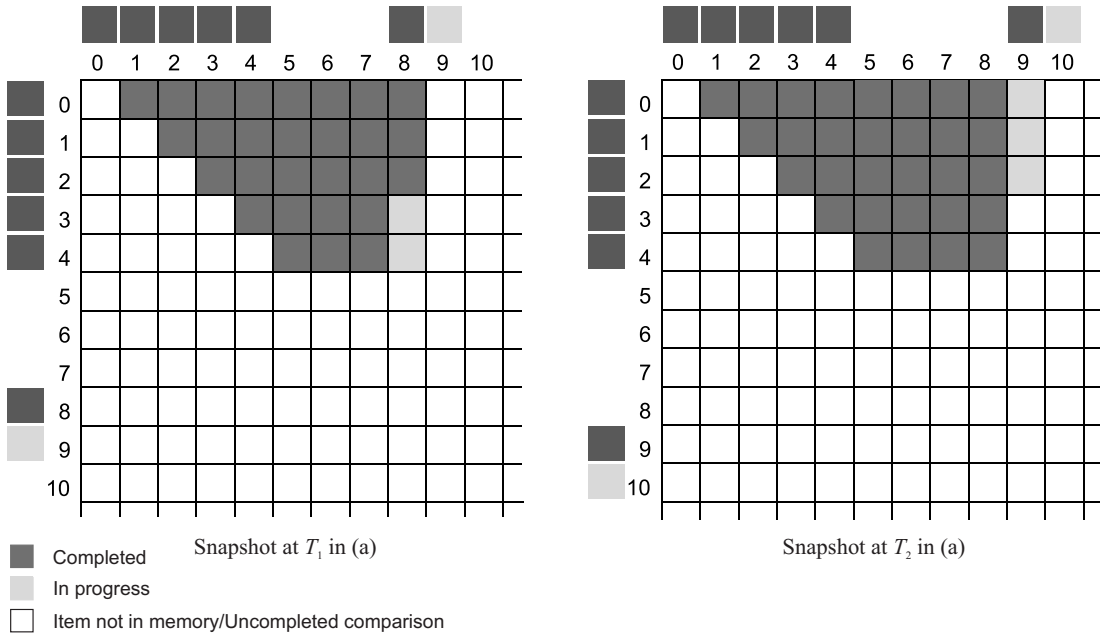
To overcome the problem of the prolonged thread waiting in Algorithm 6, the algorithm has to be improved. However, modifications need to be done without going far from its major advantage which is the lower number of *loads* it requires. In Algorithm 6, set B has room for only one data item. As a result, it is unable to *load* next data item while doing the comparisons. Therefore, increasing the size of set B to two or more to allow *loads* while doing comparisons could solve this problem.

As seen in the Figure 5.2 (a), while G_8 is compared with set A, G_9 could be loaded if there was sufficient memory remaining. Once comparisons with G_8 are completed, it can be unloaded and G_{10} can be loaded while G_9 is compared with set A. To continue this pattern the comparisons with the data item loaded first to the set B must be completed first (i.e. G_8 and then G_9, etc.). Therefore, we made the set B a queue through which the data item are pipelined. The comparisons related to the front most data item loaded to set B queue are started first. Once they are all completed, the data item is unloaded. This new extended algorithm is capable of significantly decreasing the wait time for data in some scenarios (specially with much longer *load* times than comparison times) as seen in Figure 5.3 (a) where size of the set B queue is two. The scenarios where this algorithm is efficient are discussed later in this chapter.

In Figure 5.3, only 5 comparisons are completed after each *load* to set B, compared



(a)



(b)

Figure 5.3: The matrix and thread diagrams showing the behaviour of Algorithm 6 when it is extended to use a pipe-lined set B of size two. The settings are $N = 20, M = 7$ and $P = 4$.

to 6 in Figure 5.2. This means that overall more number of repeated *loads* need to be carried out to complete a DIAC, when the size of the set B increases to two from one. Figure 5.3 (b) shows that the height of a row has decreased from 6 to 5 in compari-

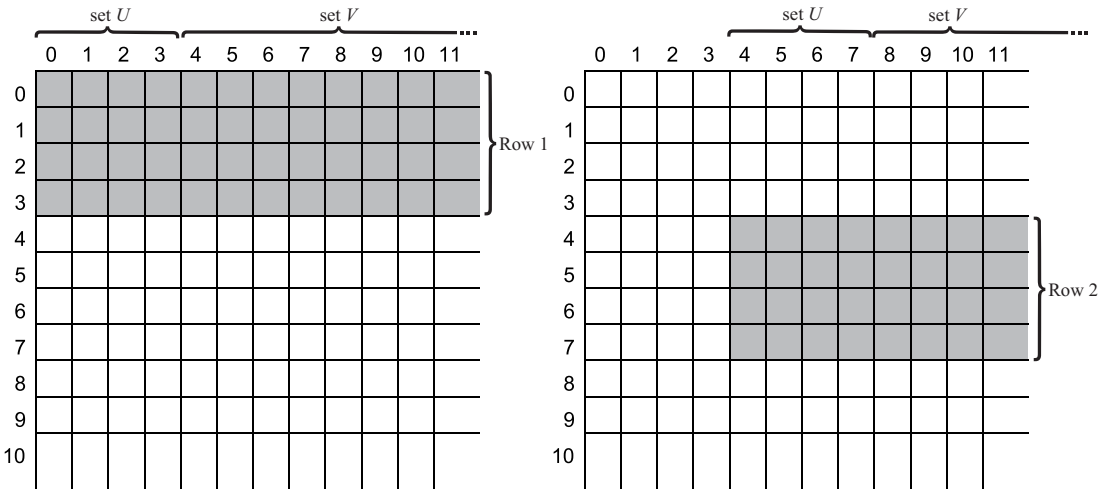


Figure 5.4: In Algorithm 7, how the correlation matrix is separated into rows and each row into two sets as set U and set V.

son with the non-extended algorithm implying that the number of rows has increased, and consequently the number of overall *loads* increases to complete all comparisons. There is always a trade-off between increasing the available tasks and minimizing the repeated *loads*. It is important to balance these two factors, for optimum results. In the next section a formal algorithm is presented using the pattern developed in this section.

5.2.2 Algorithm Development

This section develops a formal algorithm based on the *load* pattern developed in the previous section. The algorithm is depicted in Algorithm 7 but the locks used for thread-safety (synchronization) in the algorithm are not presented for simplicity. This algorithm is called "data pipeline" algorithm. It is a distributed scheduling algorithm. The main procedure COMPLETETASKS() is started by all threads in the system after the initializations of global variables from line 1 to 4. The number of threads depends on the total number of cores or processors in the system, or a number that the user prefers. This algorithm is similar to Algorithm 6, where the value of Q (size of V_{im}) is

set to one.

To describe this algorithm, each row in the correlation matrix is divided into two sections called set U and set V as seen in Figure 5.4. The set of data items remaining in the memory for the whole period of completing a row is called the set U. The rest of the data items in the row after set U is called set V. The set V_{im} is a subset of set V and only contains data items in memory. If the sizes of set U and set V are u and v respectively and the length of a row is r, v is set such that $v = r - u$. Typically, u is less than v.

Algorithm 7 creates a pipeline of data items through memory to load set V data items. The pipeline is started after loading set U to memory. The pipeline mechanism serves the two strategies addressed in Section 5.1.1.

1. The pipeline is capable of creating a continuous flow of new comparisons to memory even under limited memory.
2. By adjusting the size of the pipeline, the balance between the rates of completing and incoming comparisons can be optimized as we will discuss later.

Algorithm 7 uses a FIFO queue of comparisons, $compV$ and V_{im} to create the pipeline. The process is as follows. The data items in set V are loaded from left to right in a row (i.e. G_i with i ascending order). When a new data item in set V completes loading, the new comparisons between set U and the new item are added to the queue $compV$. The newly loaded item is added to V_{im} list for future references. The threads complete the comparisons from $compV$ in the order they were added (just before a comparison is started, it is removed from $compV$).

As a result, the first data item entered V_{im} is likely to finish all its comparisons with set U before other items in V_{im}. This happens most of the times unless a comparison takes an unusually long time to finish. Each thread checks V_{im} for data items which doesn't

Algorithm 7 Data Pipeline Algorithm: Decentralized algorithm for each thread. The algorithm is run by each thread in the system. A *load*, comparison and unload completion awakes all waiting threads. N - number of data items to be compared; M - maximum number of data item fit into memory; Q - The maximum number of data items kept in memory from set V (i.e. the maximum size of V_{im}).

```
1: Set U and V to be empty
2: V_im ← ∅ // Intialize the sub set of currently loaded items from V to be empty
3: compsU ← ∅ // Intialize list of available comparisons within U to be empty
4: compsV ← ∅ // Intialize queue of available comparisons within V ∩ U to be empty
5: procedure CompleteTasks(N, M, Q)
6:     row_start ← 0 // Intialize current row's start position to be zero
7:     row_height ← M − Q // Calculate the height of a row
8:     loadingU ← true // If true set U is being loaded
9:     InitializeSets(row_start, row_height, N)
10:    while uncompleted comparisons remaining do
11:        if max parallel loads limit reached then
12:            CompareOrWait()
13:        else
14:            if all comparisons in the current row are complete then
15:                Unload all data items in memory
16:                row_start ← row_start + row_height // Go to next row
17:                loadingU ← true // Indicate to load the next set U
18:                InitializeSets(row_start, row_height, N)
19:            end if
20:            if loadingU then
21:                item = next not loaded and not being loaded data item in U
22:                if item is the last data item in U then
23:                    loadingU ← false // Finish loading set U
24:                end if
25:                Load (item);
26:                Add comparisons between loaded data items in U and item to
   compsU
27:            else
28:                for all item in V_im do
29:                    if item has completed all comparisons with U then
30:                        Unload (item)
31:                        Remove item from V_im
32:                    end if
33:                end for
34:                item = Next never loaded data item in V for current row_start
35:                if memory is sufficient to load item then // Higher priority for loads
36:                    Load (item)
```

```
37:                     Add item to V_im
38:                     Add comparisons between item and U to compV queue
39:                 else
40:                     CompareOrWait()
41:                 end if
42:             end if
43:         end if
44:     end while
45: end procedure
46: procedure InitializeSets(row_start, row_height, N)
47:     U ← {G_x : x ∈ ℤ, row_start ≤ x ≤ row_start + row_height}
48:     V ← {G_x : x ∈ ℤ, row_start + row_height < x < N}
49:     Set V_im, compsU, compsV to be empty
50: end procedure
51: procedure CompareOrWait
52:     if compV is not empty then  // Higher priority for comparison from V
53:         comp = front most comparison in compsV queue
54:         Remove comp from compsV
55:         Complete (comp)
56:     else if compU is not empty then
57:         comp =A comparison in compsU list
58:         Remove comp from compsU
59:         Complete (comp)
60:     else
61:         Wait until notified by another thread on a system state change
62:     end if
63: end procedure
```

have any pending comparisons to be completed with set U and if found they will be unloaded to make room for new data items. When a data item in set V is unloaded, the *load* for the next data item of set V is started by a thread while other threads complete comparisons of previously loaded data items. In this way, the data items are pipelined through memory while some threads are working on comparisons and other threads constantly looking to keep the flow of new comparisons to memory.

This mechanism can also minimize the effect of fluctuations in runtime of comparisons and *loads*. Assume that a *load* takes longer than usual to complete. Instead of waiting for the struggling *load* to finish, a successor data item in set V which finished loading

early can add its comparisons to $compV$ to be completed. As soon as the comparisons are completed with the successor data item, it can be unloaded allowing a new data item to be loaded. Therefore, even if the data item are started to *load* following the order of left to right, the order in which they finish loading does not confuse or block the flow of new comparisons brought into memory. In addition, a comparison which takes longer than usual to finish does not hold back other successor comparisons from finishing, since a comparison is removed from $compV$ just before it is started. The data item related to the struggling comparison has to wait in the V_{im} until the comparison is finished. The adverse effect of this waiting is minimized by the successor data items loaded to memory which can manage to prevent the flow of incoming comparisons from completely blocking.

Algorithm 7 has a maximum size for V_{im} (Q). The size of set U which is also the height of a row is calculated as $M - Q$, where M is the maximum number of data items which can be loaded to memory. Therefore, there is room for Q number of data items from set V to be loaded into memory. The memory is allocated for a *load* as soon as the load started. So, the sum of loaded and *load* in progress data items must be less than or equal to Q.

In the algorithm, the decision making process runs repeatedly within the loop at line 10, until all comparisons in the DIAC are completed. Since the *loads* are the bottleneck in many situations, the algorithm gives priority to *loads*. Completion of a *load* on time is always important to continue the flow of bringing comparisons. Therefore, the algorithm initiates a comparison only if no *load* is possible. The condition at line 11 and 35 serves this purpose.

If the condition at line 11 is satisfied, the procedure COMPAREORWAIT() is activated. The procedure starts the front most comparison from $compsV$ queue, if it is not empty. Priority is given to the comparisons related to the data items in set V. The reason is that the comparisons within set U are always available throughout the completion of

a row, and they can be completed at any-time during the completion of the row. As a result, whenever a thread cannot find data items to *load* or a comparison from set V, it can complete a comparison within set U. So, if $queueV$ is empty and $compsU$ is not empty, a thread will start a comparison from $compsU$. It can be argued that delaying comparisons in $compsU$ may later become a hindrance to finishing the row. However, the threads waiting for data items in set V to be loaded can instead complete comparisons from $compsU$. This is more effective than finishing them first as we will see in Section 5.3.1. On the other hand, regardless of the position where the comparisons in $compsU$ are completed, the same amount of computational time is needed for them and so completing them, when threads could have waited otherwise, is always effective.

If neither *load* nor a comparison is possible, the thread will wait until the system status changes. A system state change occurs in the following three scenarios.

1. A comparison finishes and one of the data items in the comparison becomes available to unload.
2. A *load* completes.
3. An unload occurs.

When one of above events occurs all waiting threads are notified to resume.

Once all comparisons in the row are completed the next row must begin. This is done inside the condition at line 14. This code is executed once by a thread at the completion of all comparisons in a row. After moving to the next row, all data items in memory are unloaded and all queues and lists are cleared. The flag indicating that set U is being loaded is set to true since next row's set U must be loaded next.

Inside the condition at line 20, the current set U is loaded. If the maximum number of parallel loads permits, every thread is allowed to start a *load*, since memory is sufficient

to load all data items in set U. It is important to note that if the threads have reached the maximum number of allowed parallel *loads*, the condition at line 11 will prevent any more threads from starting another *load*. Once the last data item in set U is started loading, the flag $loadingU$ is set to $false$ indicating that the set U is completely loaded or being loaded, and set V must be loaded here onwards. Every time a *load* of a data item in set U is completed, the comparisons it brings with currently loaded data items in set U are added to $compsU$ at line 26. This ensures that whenever a thread is unable to start a *load*, it has comparisons to complete in $compsU$ list.

The code starting from line 27 (*else* of the condition at line 20), is responsible for loading and unloading process of set V. First it checks whether any data item in V_{im} has completed all comparisons with set U. If so, the data item is unloaded to allow another *load*. Then it checks if memory is sufficient to load the next data item in set V. If memory is sufficient, the data item is loaded.

At line 35, if memory is sufficient to load the next data item from set V, it will be loaded. Otherwise, a process similar to the one occurs when no *load* is possible due to reaching the maximum parallel loads will be activated (i.e. call to COMPAREORWAIT()). At the end of this process, a comparison will be chosen to be started if possible or the thread will wait until the next system change.

5.3 Analysis of the Algorithm Behaviour

This section discusses a few examples to give an insight into the behaviour of the algorithm. It also discusses the factors affecting the behaviour of the algorithm and their influence on the algorithm behaviour.

5.3.1 Examples for Understanding the Behaviour of the Algorithm

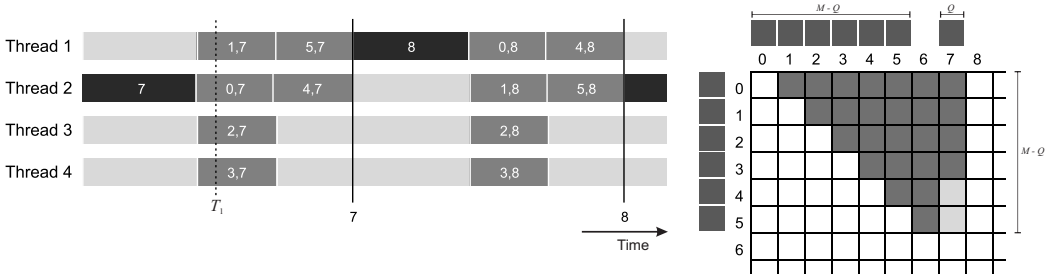
Before deeply discussing the factors affecting the behaviour of the algorithm, it is important to have a good understanding of the behaviour of the algorithm. We will consider the most common scenario where parallel *loads* are inefficient (when data items are read from a sequential read disk). For the convenience of presentation, small values are used for N, P and M in the figures.

Figure 5.5 shows the behaviour of the algorithm when Q, the size of V_{im} changes:

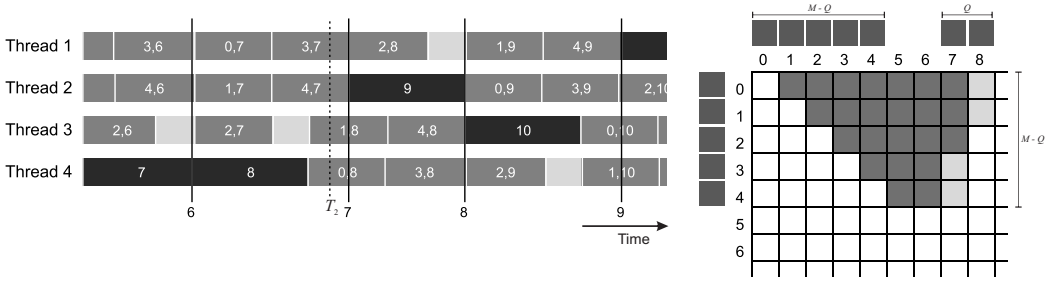
- Figure 5.5 (a) shows that when $Q = 1$, all threads wait until a data item is loaded to the memory.
- When Q increases to 2 the wait time of the threads decrease as seen in Figure 5.5 (b) because the data is starting to pipeline through V_{im}.
- When Q further increases to 3, the wait time decreases slightly more as seen in Figure 5.5 (c).

It is noted that when Q is three, the *loads* are continuously carried out. The reason is that at any given time there is a free memory slot to *load* the next set V item, since comparisons with a previous data item can be completed and unloaded, before V_{im} becomes full. Therefore, the rate of loading data items has reached its maximum. When Q is increases to 4, the wait times increases as seen in Figure 5.5 (d). For the configurations shown in Figure 5.5, scenario $Q = 2$ is better than other Q values.

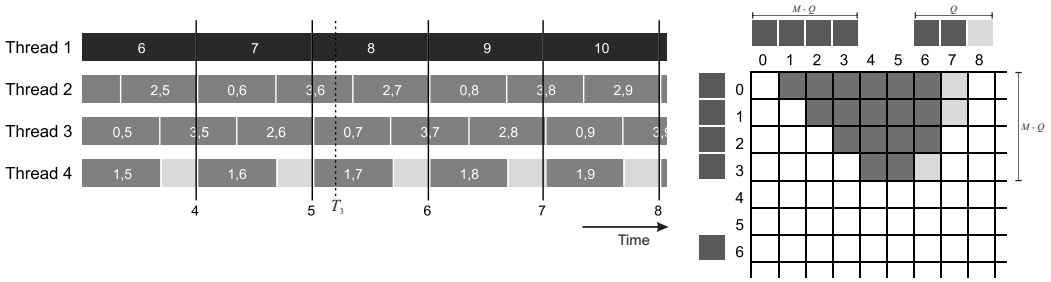
The criterion to choose the better Q for the row is the rate of completing comparisons (i.e. number of comparisons completed in the row divided by the time spent to complete the row). Even though $Q = 3$ still has slightly less wait times in threads, $Q = 2$ has a higher rate of completing comparisons in the row. The reason is that the reduction in wait time when $Q = 3$ is insufficient to compensate for the increased number of repeated *loads* because of the decreased row height. It is noted that increasing Q has



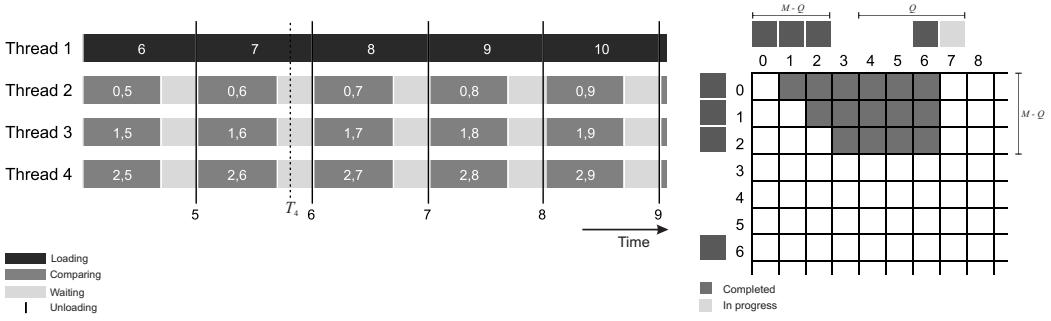
(a) $Q = 1$; Matrix diagram snapshot at T_1



(b) $Q = 2$; Matrix diagram snapshot at T_2



(c) $Q = 3$; Matrix diagram snapshot at T_3



(d) $Q = 4$; Matrix diagram snapshot at T_4

Figure 5.5: The matrix and thread diagrams showing the behaviour of Algorithm 7 with $Q = 1$ (a), 2 (b), 3 (c) and 4 (d). The settings are $N = 20, M = 7$ and $P = 4$. The maximum number of parallel *loads* is limited to one.

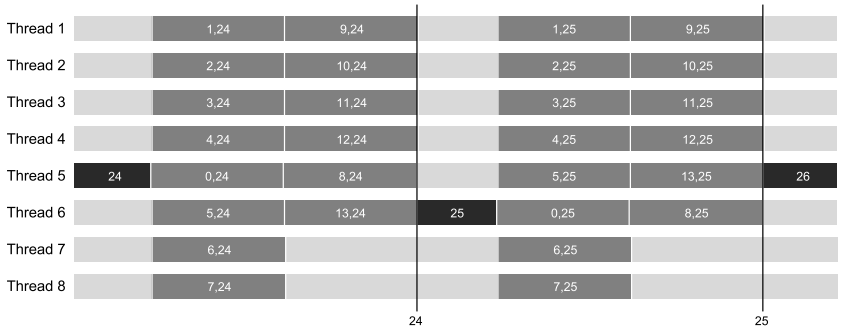
an adverse effect of increasing the repeated *loads*. Methods to pre-determine optimum Q will be discussed later in this section.

Figure 5.6 shows another scenario where relatively high number of data items can be held in memory compared to the scenario shown in Figure 5.5. As seen in Figure 5.6 (a) when $Q = 1$ some threads still wait for loading data. But when Q is set to two, no thread is waiting for data or comparisons as seen in Figure 5.6 (b). Every thread is working continuously in this situation.

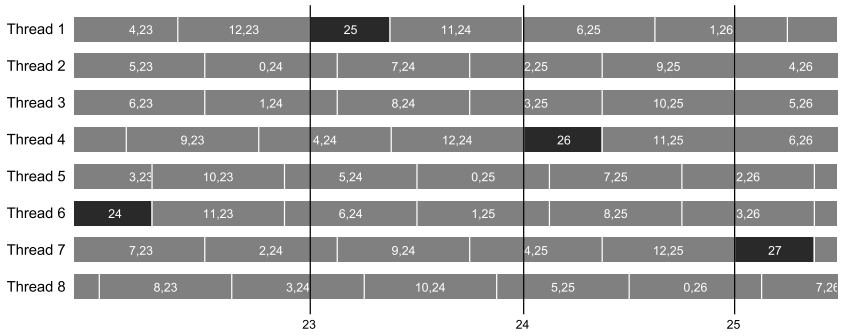
Importantly, in Figure 5.6 (b) there is no clear pattern in the order of completing comparisons explicitly specified in Algorithm 7. Since the threads are allowed to make decisions independently, Algorithm 7 is designed in a way that a proper order of completing comparisons is gradually formed among threads which eventually keeps the data pipeline flowing continuously.

When Q increases to 3, there is not much difference in the thread diagram shown in Figure 5.6 (c), since all threads are busy continuously same as Figure 5.6 (b). However, it is important to look at the behaviour of this scenario in the grid diagrams shown in Figure 5.7. The figure shows three snapshots of three critical places. In figures 5.7 (a) and (b), G_{16} is not used when it is at both left and middle positions in V_{im}. Then, G_{17} is not being used in Figure 5.7 (c) even if it is loaded and ready to be used. The meaning of 'using' a data item is that the comparisons related to the data item are currently being performed. Once all threads become busy with work at a value of Q (at $Q = 2$ in this case), increasing Q will not create further gain. The extra data items in V_{im} will be unused and cause more redundant repeated *loads*. As a result, $Q = 2$ is the best Q in this scenario.

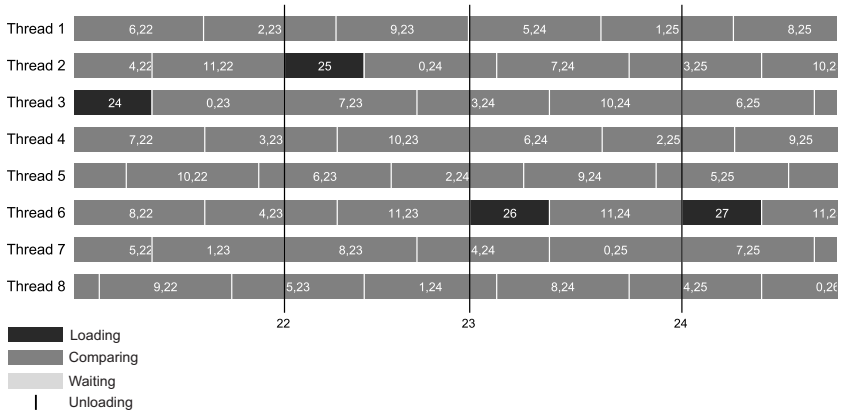
The examples so far in this section produce schedules with prolonged wait times if $Q = 1$. However, there are two scenarios where $Q = 1$ is better than other Q values in Algorithm 7 (a theoretical analyses on this will be conducted in Section 5.4).



(a) $Q = 1$



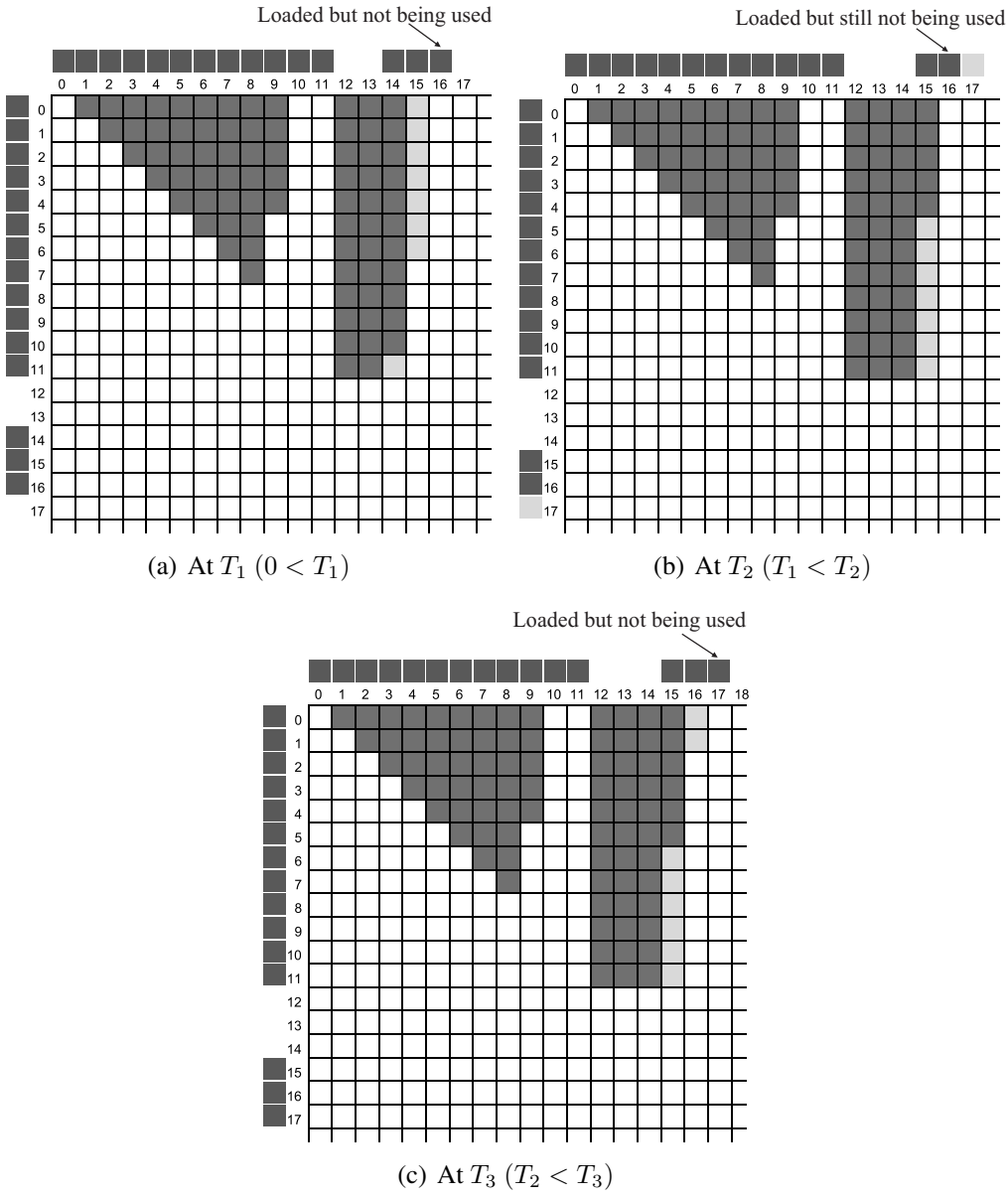
(b) $Q = 2$



(c) $Q = 3$

Figure 5.6: The thread diagrams showing the behaviour of Algorithm 7 when Q is 1 (a), 2 (b) and 3 (c). The settings are $N = 50, M = 15$ and $P = 8$. The maximum number of parallel *loads* is limited to one.

- The first scenario is when *load* time is extremely low compared to the comparison times and as a result the wait time for data is significantly low.



(a) At T_1 $(0 < T_1)$

(b) At T_2 $(T_1 < T_2)$

(c) At T_3 $(T_2 < T_3)$

Figure 5.7: Few snapshots of the grid diagram showing the behaviour of Algorithm 7 when $Q = 3$. The settings are $N = 50, M = 15$ and $P = 8$ same as Figure 5.6. The maximum number of parallel *loads* is limited to one.

- The second scenarios is when the comparisons within set U cannot be completed at the same rate as they are bought in.

The second scenario is shown in Figure 5.8. The first diagram in Figure 5.8 (a) shows

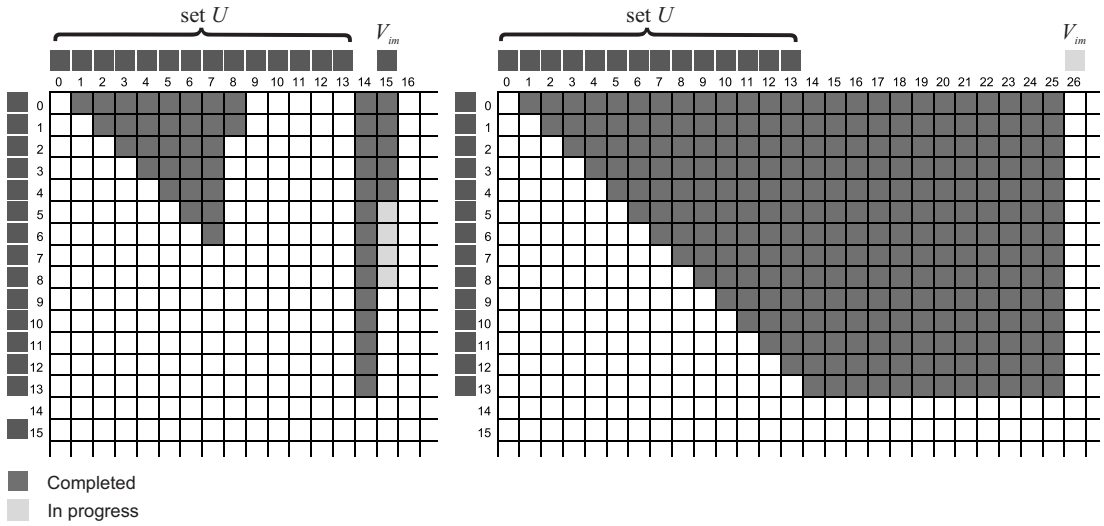
that there are still uncompleted comparison within set U when loading set U is finished. When there are comparisons remaining within set U, whenever a thread has to wait for a *load* of set V, it can start a remaining comparison from set U. This can be seen in Figure 5.8 (b) where comparisons from set U are completed in between each *load* of set V. However, the comparison within set U are all completed when G_{26} of set V is loaded. Thereafter, the threads have to wait for *loads* from set V similar to as seen previously in Figure 5.6 (a). In some scenarios where there large number of uncompleted comparisons remain just after loading full set U, the threads do not wait for data in a major portion of a row even if $Q = 1$, where it could be better than $Q = 2$.

5.3.2 Factors Affecting the Behaviour of the Algorithm

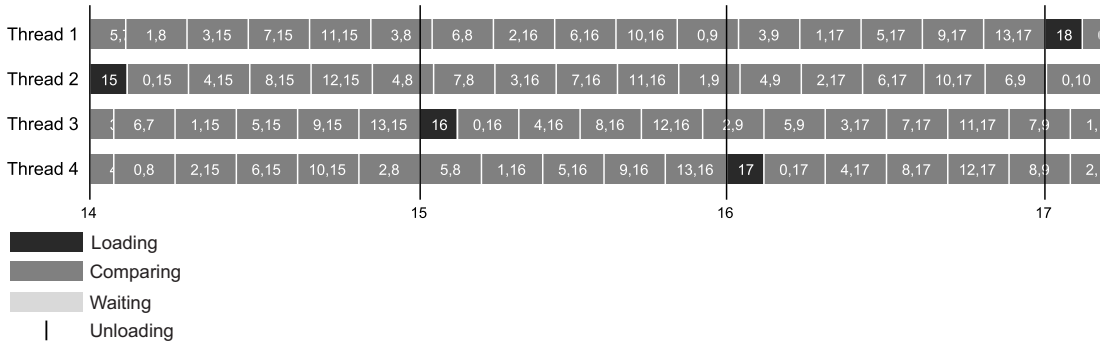
After looking at the behaviour of the "data pipeline" algorithm (Algorithm 7), this section discusses the factors that affect the behaviour of the algorithm. There are four such factors:

1. the ratio between *load* time and comparison time (i.e. t_c/t_l where t_c is the time required for a comparison and t_l is the time required for a *load*);
2. the maximum number of data items that can be held in memory (M);
3. the number of threads (P); and
4. the maximum number of parallel loads (L_P).

The behaviour of the algorithm mainly depends on the rate of completing work and rate of introducing work. The 'work' means the time to complete comparisons. For example, if a *load* brings in n comparisons, it brings $n \times t_c$ work. When the ratio t_c/t_l is higher, a single *load* brings more work ($n \times t_c$) within less time (t_l). Only the ratio between these two affects the behaviour of the algorithm, not their actual values.



(a) Snapshots at T_1 (left) and T_2 (right); $T_1 < T_2$



(b) In between T_1 and T_2 in Figure (a) above

Figure 5.8: The snapshots of the grid and thread diagrams showing the behaviour of Algorithm 7 when $Q = 1$ and it is still effective. The settings are $N = 50, M = 15$ and $P = 4$. The maximum number of parallel *loads* is limited to one.

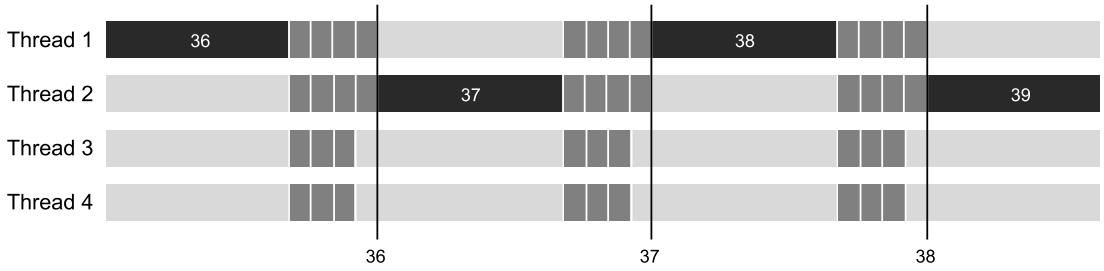
When M increases, the number of comparisons which a *load* can bring in increases. This makes the threads do more work (comparisons) for a smaller number of *loads*. So, the increased number of incoming comparisons eventually affects the behaviour of the algorithm. Often, with higher M values, smaller Q values tend to produce better schedules, since the rate of introducing work is high when M is large.

When the number of cores, P increases, the system's maximum capable rate of completing comparisons increases. Therefore, to produce better schedules, the rate of bringing comparisons must also be increased. For higher P vales the algorithm has to settle for a higher Q values more often to make more work available to threads. Even if the height of the row decreases when Q increases, the number of comparisons available can be increased up to certain limit by increasing Q, as we will see in the experiments.

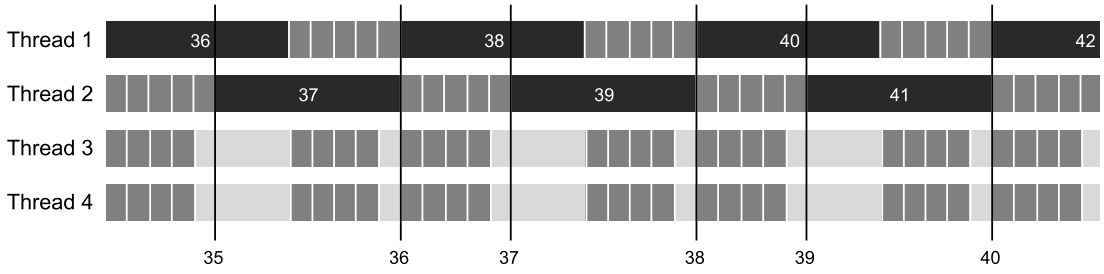
Similar to M, L_P has the potential of increasing the rate of introducing comparisons. Whenever the rate of completing comparisons is ahead of introducing comparisons, increasing Q will provide an opportunity for parallel loads to supplement the rate of bringing comparisons. A scenario that allows more than one parallel *loads* is shown in Figure 5.9. As seen in Figure 5.9 (b), when $Q = 2$ the rate of bringing comparisons does not match the rate of completing comparisons. In comparison, when Q is set to $Q = 3$ in Figure 5.9 (c), no thread is waiting. In this case the ability to do *load* in parallel makes all threads busy by matching the rate of bringing comparisons to the rate completing comparisons. $Q = 3$ is optimum in this case, because there is no further gain for increasing Q after all threads become continuously busy.

5.4 Determination of Algorithm Parameters

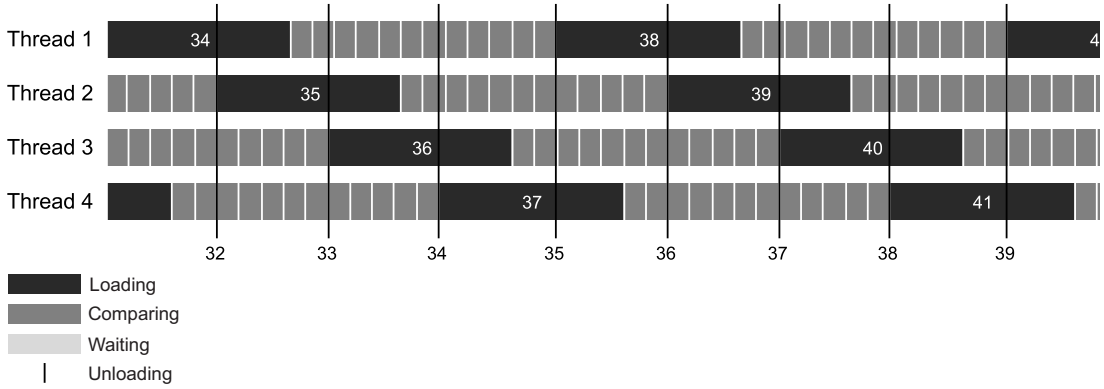
It is important to determine the proper parameter settings for Algorithm 7 to ensure proper load balancing among the threads for optimum performance. While examining the behaviour of Algorithm 7 in Sections 5.2, it is noted that predicting the pattern in which the threads carry out comparisons is difficult. Therefore, mathematical modelling to find optimum parameter settings becomes extremely complex in certain scenarios, especially when parallel loads are possible. As a result, we limit our mathematical modelling only to the systems in which only single parallel *load* is efficient.



(a) $Q = 1$



(b) $Q = 2$



(c) $Q = 3$

Figure 5.9: The snapshots of the thread diagrams showing the behaviour of Algorithm 7 (a) $Q = 1$, (b) $Q = 2$ and $Q = 3$. The other settings are $N = 50, M = 15$ and $P = 4$. The maximum number of parallel *loads* is limited to **two**.

Such systems are most commonly found. For other scenarios, we introduce a fast simulation technique to determine the parameter settings.

5.4.1 Maximum Size of V_{im} (Q) in Algorithm 7

In Algorithm 7, the size of Q is the only adjustable parameter. Even though it uses a similar Q size for each row for simplicity, optimum way would be to dynamically determine Q for each row. Let Q_{opt} denote the best Q value for a row. It is important to determine an upper bound for Q_{opt} to help determining Q_{opt} by trying different Q values in simulations. This section develops a method to decide an upper bound for Q_{opt} for a row which is called Q_{max}. Q_{max} is defined so that the best Q, Q_{opt} is $1 \leq Q_{opt} \leq Q_{max}$.

Q_{max} depends directly on the maximum efficient parallel *loads*, L_P. The reason is that the only purpose of increasing Q is to encourage *loads*. When Q is high, there is room for more parallel *loads* in memory. Therefore, we develop a theory for calculating Q_{max} in this section.

Theorem 11. *In Algorithm 7, let Q_{opt} denote the best Q for a row, Q_{max} denote the upper bound of Q_{opt} (i.e. $1 \leq Q_{opt} \leq Q_{max}$) and L_P denote the maximum number of efficient parallel loads. If $L_P = 1$ and either Equation (5.1) or Equation (5.2) is satisfied, then $Q_{max} = 2$.*

$$t_l \geq \left\lceil \frac{M-2}{P-1} \right\rceil t_c \tag{5.1}$$

$$t_l \leq \left\lfloor \frac{M-2}{P-1} \right\rfloor t_c. \tag{5.2}$$

Proof. When Equation (5.1) is satisfied, the system is capable to finish all comparisons with the last loaded data item to V_{im} (G_x) within t_l (load time of a data item) as seen in Figure 5.10. In this case, the *load* of the next data item (G_y) to V_{im} can be started as soon as G_x finishes. When this happens, the system is carrying out *loads* continuously. Therefore, even if Q is increased further, there is no opportunity to increase the rate of completing *loads*. The system has reached its maximum rate of completing comparisons. Therefore, $Q_{max} = 2$ in this scenario.

When Equation (5.2) is satisfied, the system is unable to finish all comparisons with G_x within t_l, it must wait for the comparisons to finish before starting to *load* G_y. In this case, increasing Q over two cannot increase rate of completing comparisons since the system cannot handle more comparisons than it is already receiving within t_l. Increasing Q will only decrease the rate of completing comparisons because it will decrease the size of set U ($M - Q$) and decrease the number of comparisons a single *load* brings in. It can be argued that when Q is increased largely over 2, the system will be able to finish comparisons more quicker because of the decrease of comparisons a single *load* brings in, which might eventually increase the rate of loading up to the rate which occurs when loads are carried out continuously. However, as we will see in the following proof, increasing Q over 2 decreases the rate of completing comparisons.

In Algorithm 7, to move to the comparisons with the next data item in V_{im}, all available comparisons with the previously loaded data item must be started (and completed to threads to become free). Therefore, if all comparisons brought in by a data item cannot be completed within t_l (i.e. Equation (5.2) is satisfied), the system will have to wait for them to finish, before it starts the next set of comparisons with the newly loaded data items, even though the later comparisons are already in memory. When $Q \geq 2$, the time taken over t_l to finish all comparisons brought by a *load* is denoted by $t_x > 0$. While a *load* is in progress, one thread is busy with the *load* and the rest of the threads carry out comparisons. Once the *load* is done all threads finish the comparisons. Hence, when $Q \geq 2$,

$$\underbrace{t_c(M-Q)}_{\text{Work brought in by a } load} - \underbrace{t_l(P-1)}_{\text{Work completed within } t_l} = \underbrace{Pt_x}_{\text{Remainder of the work after } t_l} \tag{5.3}$$

If the rate of completing comparisons is R, then

$$R = \frac{(M-Q)}{t_l + t_x}. \tag{5.4}$$

By applying t_x from Equation (5.3) to Equation (5.4) we get:

$$R = \frac{P(M-Q)}{t_l + t_c(M-Q)}. \tag{5.5}$$

Let q be a continuous variable (over the range $2 \ldots M-1$) that equates with Q at discrete points. To decide the behaviour of R versus Q, R is differentiated against q. Note that in Equation (5.3), Q is not a continuous variable, so we cannot differentiate R with respect to Q.

$$\frac{dR}{dq} = \frac{-t_l \cdot P}{(t_l + t_c(M-q))^2} \tag{5.6}$$

$\frac{dR}{dq} < 0$ is always satisfied according to Equation (5.6). Therefore, when Q is increased beyond 2 the rate of completing comparisons, R decreases if the system is unable to complete all comparisons within t_l time (i.e. when Equation (5.2) is satisfied). Therefore, Q_{max} is two for when Equation (5.2) is satisfied as well. This completes the proof. □

The results of Equation (5.6) do not mean that increasing Q will always decrease the rate of completing comparisons. According to the conditions of Equation (5.3) it means that once all threads are continuously busy with work, increasing Q will reduce the rate of completing comparisons. In the next theorem, Theorem 11 is extended to address L_P values greater than one.

Theorem 12. *In Algorithm 7, let Q_{opt} denote the best Q for a row, Q_{max} denote the upper bound of Q_{opt} (i.e. $1 \leq Q_{opt} \leq Q_{max}$) and L_P denote the maximum number of efficient parallel loads. If either Equation (5.1) or Equation (5.2) is satisfied then $Q_{max} = 2L_P$.*

Proof. When $L_P > 1$ every single channel for loading can be considered separately as

when $L_P = 1$ and $Q = 2$. Therefore, the system reaches is full potential for completing comparisons either by:

- continuously loading data items in every channel or
- not being able complete the comparisons at the rate they are brought in;

when each channel has its own extra slot in memory to continue *loads* according to Theorem 11. When two slots of memory is allocated for each channel, the *loads* can continuously flow through each channel if the system is capable of completing comparisons at this rate. Therefore, Q_{max} is $Q_{max} \leq 2L_P$. This completes the proof. □

Theorem 11 and Theorem 12 are experimentally validated in Section 5.4.4.

5.4.2 Optimum Size of V_{im} (Q) in Algorithm 7

As we have already mentioned, when L_P is higher than one, determining the optimum queue size becomes increasingly complex. This section develops a mathematical solution to determine the optimum Q value, Q_{opt} when $L_P = 1$, which is the most common scenario for most of the systems. According to Theorem 11, the optimum Q could be either 1 or 2 for $L_P = 1$. Therefore, our approach is to develop mathematical equations to estimate the runtime of each row when $Q = 1$ and $Q = 2$ and determine Q_{opt} by the highest rate to completing comparisons for the row.

The mathematical modelling is divided into scenarios $t_l \geq \left\lceil \frac{M-2}{P-1} \right\rceil t_c$ and $t_l \leq \left\lfloor \frac{M-2}{P-1} \right\rfloor t_c$. Each of the two scenarios has a clearly different behaviour in the runtime based on the properties of the system and the dataset.

Scenario 1:

In Scenario 1, Equation (5.1) is satisfied. Figure 5.10 shows an instance of this scenario. As seen in the Figure 5.10, the threads can finish the maximum number of comparisons brought by a single load ($M - Q$) within t_l, causing following two phenomena to occur.

- All comparisons within set U are compared as soon as set U finishes loading.
- When $Q = 2$, the next *load* starts as soon as the previous *load* finishes throughout the row (i.e. *loads* are carried out continuously).

For this scenario we develop the following theorem to decide the optimum value of Q.

Theorem 13. *If Equation (5.1) is satisfied and $L_P = 1$, the optimum Q, Q_{opt}, is derived such that:*

$$Q_{opt} \begin{cases} = 1; & R'_{S1} \geq R''_{S1} \\ = 2; & R'_{S1} < R''_{S1} \end{cases} \tag{5.7}$$

where

$$R'_{S1} = \frac{(M-1)(2r_i - M)}{2\left[t_l(M-1) + (r_i - M + 1)\left(\left\lceil \frac{M-1}{P} \right\rceil t_c + t_l\right)\right]}, \tag{5.8}$$

and

$$R''_{S1} = \frac{(M-2)(2r_i - M + 1)}{2\left[t_l r_i + \left\lceil \frac{M-2}{P} \right\rceil t_c\right]}, \tag{5.9}$$

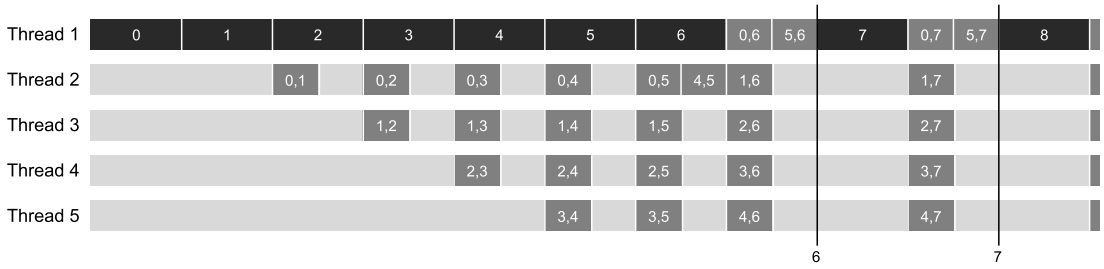
where r_i denotes the length of the current row (i.e. remaining data items to be compared).

Proof. When $Q = 1$, the time taken to complete the row, t'_{s1}: (see Figure 5.10 (a)):

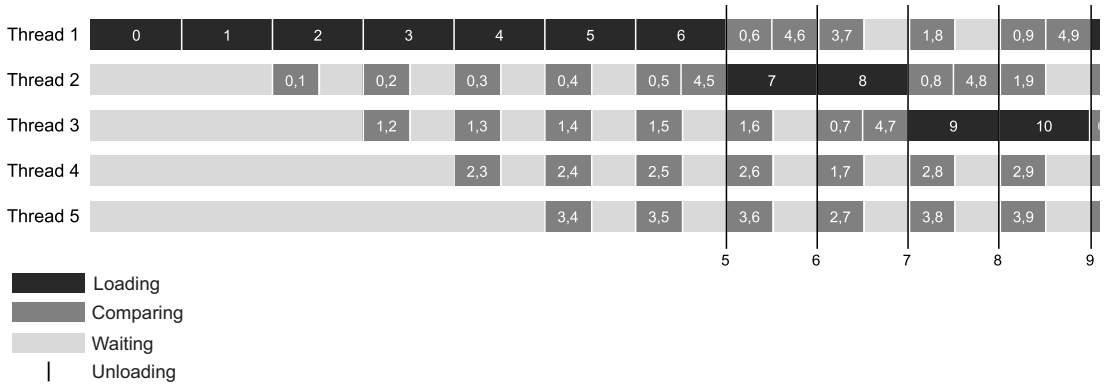
$$t'_{s1} = \underbrace{t_l(M-1)}_{\text{Loading set } U} + \underbrace{(r_i - M + 1)}_{\text{Number of set } V \text{ loads}} \underbrace{\left(\left\lceil \frac{M-1}{P} \right\rceil t_c + t_l\right)}_{\text{Loading data item to set } V \text{ and completing its comparsons}}$$

Let n_{q1} denote the number of comparisons in a row when $Q = 1$, we have

$$n_{q1} = \frac{1}{2}(M-1)(2r_i - M)$$



(a) $Q = 1$



(b) $Q = 2$

Figure 5.10: The snapshots of the thread diagrams showing the behaviour of Algorithm 7 when Equation (5.1) is satisfied (Scenario 1) (a) $Q = 1$ and (b) $Q = 2$. The settings are $N = 50, M = 15, P = 5, t_c = 50s$ and $t_l = 100s$. The maximum number of parallel *loads* is limited to one.

Therefore, the rate of completing comparisons for the row is R'_{S1}:

$$R'_{S1} = \frac{n_{q1}}{t'_{s1}} \tag{5.10}$$

When $Q = 2$, the time taken to complete the row, t''_{s1}: (please refer Figure 5.10 (b)):

$$t''_{s1} = \underbrace{t_l r_i}_{\text{Contineously loading all data items in the row}} + \underbrace{\left\lceil \frac{M-2}{P} \right\rceil t_c}_{\text{Completing comparisons with the last data item in the row}}$$

Let n_{q2} denote the number of comparisons in a row when $Q = 2$

$$n_{q2} = \frac{1}{2}(M-2)(2r_i - M + 1). \tag{5.11}$$

Therefore, the rate of completing comparison for the row is R''_{S1}:

$$R''_{S1} = \frac{n_{q2}}{t''_{s1}} \tag{5.12}$$

If $R'_{S1} \geq R''_{S1}$, $Q = 1$ has higher or equal rate of completing comparisons compared to $Q = 2$. When $R'_{S1} = R''_{S1}$, $Q = 1$ has a slight edge over $Q = 2$ because of the lower complexity in the scheduling algorithm for lower Q values. Therefore, if $R'_{S1} \geq R''_{S1}$, $Q_{opt} = 1$ for the row. Otherwise, Q_{opt} for the row is $Q = 2$. This gives Equation (5.7). Equation (5.10) and Equation (5.12) give Equation (5.8) and Equation (5.9) respectively. This completes the proof. □

Scenario 2:

Scenario 2 is when the Equation (5.2) is satisfied. An instance of this scenario is depicted in Figure 5.11. In this scenario, the maximum number of comparisons a *load* brings $(M - Q)$ cannot be completed within t_l. Therefore, three phenomena occur.

- All comparisons within set U cannot be completed while loading set U.
- When $Q = 2$, every thread is busy after loading set U.
- When $Q = 1$, the comparisons remaining from set U are used to fill the gaps when threads are waiting for set V data items to load.

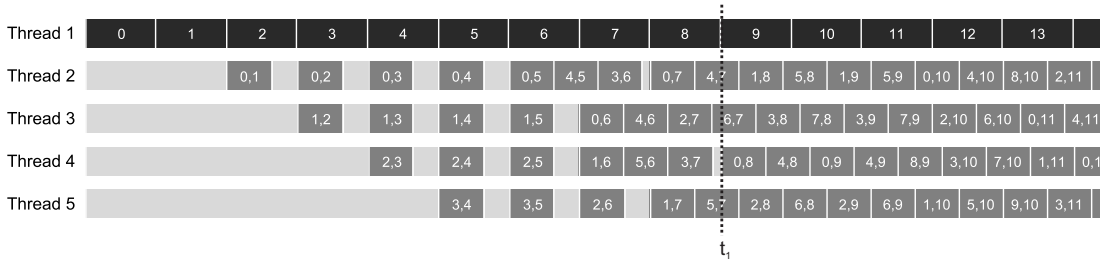
For this scenario, we have the following theorem for Q_{opt}.

Theorem 14. *If Equation (5.2) is satisfied and $L_P = 1$, the optimum Q, Q_{opt} is*

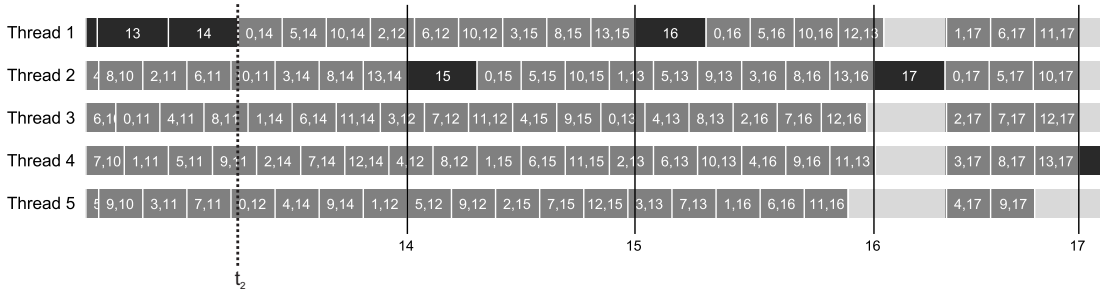
$$Q_{opt} \begin{cases} = 1; & W_{wait} \leq W_{rem} \\ = 1; & W_{wait} > W_{rem} \textit{ and } R'_{S2} \geq R''_{S2} \\ = 2; & W_{wait} > W_{rem} \textit{ and } R'_{S2} < R''_{S2} \end{cases} \tag{5.13}$$

where

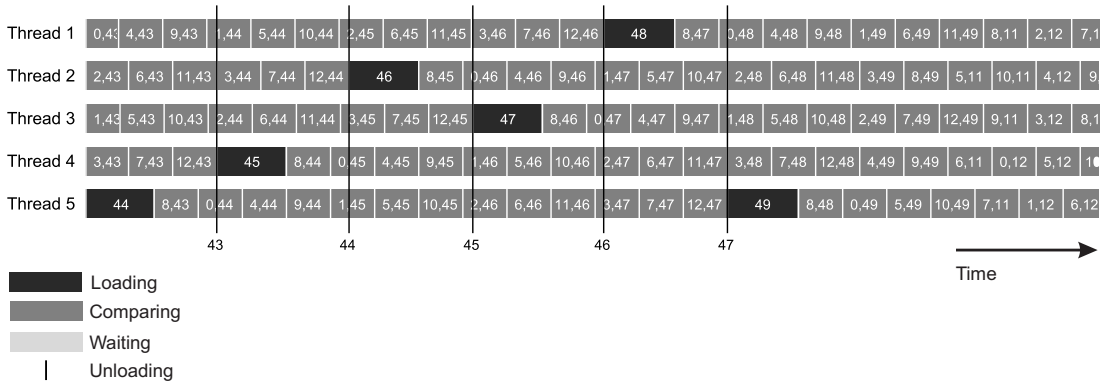
$$W_{wait} = (M - N + 2)\left[t_l(P-1) + t_c\left\lceil\frac{M-1}{P}\right\rceil - t_c(M-1)\right], \tag{5.14}$$



(a) $Q = 1$; while loading set U



(b) $Q = 1$; after loading set U



(c) $Q = 2$; after loading set U

Figure 5.11: Snapshots of the thread diagrams showing the behaviour of Algorithm 7 when Equation (5.2) is satisfied (Scenario 2). Sub-figure (a) and (b) shows two sections of the same thread graph when $Q = 1$ and sub-figure (c) shows $Q = 2$. The other settings are $N = 50, M = 15, P = 5, t_c = 50s$ and $t_l = 80s$. The maximum number of parallel *loads* is limited to one.

$$W_{rem} = \frac{Mt_c(M-1) - (P-1)(2t_lM - 2t_l - t_1)}{2}, \tag{5.15}$$

$$R'_{S2} = \frac{(M-1)(2r_i - M)}{2\left[t_l(M-1) + (r_i - M + 1)\left(\left\lceil \frac{M-1}{P} \right\rceil t_c + t_l\right)\right]}, \tag{5.16}$$

and

$$R''_{S1} = \frac{P(M-2)(2r_i - M + 1)}{2r_i t_l + (P-1)(t_1 + 2t_l) + t_c(M-2)(2r_i - M + 1)}, \tag{5.17}$$

where r_i denotes the length of the current row (i.e. remaining data items to be compared). t_1 is defined as:

$$t_1 = \frac{t_l^2}{t_c}(P-1) + t_l. \tag{5.18}$$

Proof. Let t_1 be the time when all threads become busy while loading set U, as seen in Figure 5.11 (a). The value of t_1 is calculated by using the following method. The rate of bringing work is calculated first while loading set U. When the rate is equal to the system's capacity to complete comparisons, all threads becomes busy at the estimated value of t_1. After loading the first data item in the row, each i^{th} load brings i number of comparisons. If $t = 0$ when the row begins, after $t = t_l$ at each $t = it_l$, it_c work is brought in. Since the time for a *load* is t_l and $i = 0$ at $t = t_l$,

$$i = \frac{t - t_l}{t_l}. \tag{5.19}$$

At i, the rate of bringing in comparisons for set U is denoted as R_a. We have

$$R_a = \frac{it_c}{t_l} \tag{5.20}$$

Substituting Equation (5.19) to Equation (5.20) yields

$$R_a = \frac{t_c}{t_l^2}t - \frac{t_c}{t_l}. \tag{5.21}$$

Since the *loads* are performed continuously by a thread while loading set U, the system is capable of completing comparisons at the rate of $P - 1$. Therefore, $t = t_1$ when $R_a = P - 1$. By solving $R_a = P - 1$ using Equation (5.21) we get Equation (5.18).

When $Q = 1$, while loading set V, the threads wait for other threads to complete their comparisons and to *load* next set V data item, as seen in Figure 5.11 (b). If this wait

time can be completely filled with the remaining work from comparisons within set U, $Q = 1$ is the best Q value since $Q = 1$ does more comparisons in a row. If a schedule with $Q = 1$ does not have any thread waiting times after loading set U, it does more comparisons with fewer *loads* (at a higher rate than $Q = 2$). Therefore, we first look for the scenario where $Q = 1$ and the thread wait time, W_{wait} is less than the remaining work from set U, W_{rem} (i.e. $W_{wait} \leq W_{rem}$).

$$W_{wait} = \underbrace{(M - N + 2)}_{\text{Repetitions of the pattern}} \left[\underbrace{t_l(P-1)}_{\text{Wait time for } \textit{loads}} + \underbrace{t_c \left(\left\lceil \frac{M-1}{P} \right\rceil - (M-1) \right)}_{\substack{\text{Wait time while other} \\ \text{threads completing comparisons}}} \right] \tag{5.22}$$

Simplifying Equation (5.22) gives Equation (5.14). Loading the first M data items finishes at $t = t_2$, as seen in Figure 5.11 (b).

$$W_{rem} = \underbrace{\frac{Mt_c(M-1)}{2}}_{\text{Work available till } t_2} - \left[\underbrace{\frac{(P-1)(t_1 - 2t_l)}{2}}_{\text{Work completed till } t_1} + \underbrace{(P-1)(Mt_l - t_1)}_{\text{Work completed from } t_1 \text{ to } t_2} \right] \tag{5.23}$$

Simplifying Equation (5.23) gives Equation (5.15).

Therefore, when $W_{wait} \leq W_{rem}$, $Q_{opt} = 1$. When $W_{wait} > W_{rem}$ and $Q = 1$, the wait times of threads are filled partially with W_{rem}. However, the time to complete the row is similar to Scenario 1 ($Q = 1$), since only the gaps are filled with W_{rem}. Therefore, the rate of completing comparisons when $Q = 1$ in Scenario 1, R'_{S1} is similar to R'_{S2} when $W_{wait} > W_{rem}$. So, Equation (5.10) gives Equation (5.16) for Scenario 2, when $Q = 1$.

To calculate the rate of completing comparisons when $Q = 2$, R''_{S2}, we take the following approach. In this case, all threads are continuously busy with work after t_1.

Therefore, we sum the total work including the *loads* which are carried out by the threads after t_1 and divide the sum by P to calculate the runtime of the row after t_1. If the total work completed by all threads in the row after t_1 is W_{t1}, we have

$$W_{t1} = \underbrace{\left(r_i - \frac{t_1}{t_l}\right)}_{\text{Time spent on loads after } t1} + \underbrace{t_c\left(\frac{(M-2)(M-3)}{2} + [r_i - (M-2)](M-2)\right)}_{\text{Total work from comparisons in the row}} - \underbrace{\frac{(P-1)(t_1 - 2t_l)}{2}}_{\text{Work from comparisons completed till } t_1} \tag{5.24}$$

The total completion time for the row, t''_{S2} is

$$t''_{S2} = t_1 + \frac{W_{t1}}{P} \tag{5.25}$$

Since the number of comparisons in the row, n_{q2}, can be taken from Equation (5.11), the rate of completing comparisons, R''_{S2} is:

$$R''_{S2} = \frac{n_{q2}}{t''_{S2}}. \tag{5.26}$$

Solving Equation (5.26) using Equations (5.24) and (5.25) gives Equation (5.17). When $W_{wait} > W_{rem}$ and $R'_{S2} \geq R''_{S2}$, $Q = 1$ has a higher or equal rate of completing comparisons than $Q = 2$. When the rates are equal, $Q = 1$ has a slight edge over $Q = 2$ because of the lower complexity in the scheduling algorithm for lower Q values. Therefore, if $R'_{S2} \geq R''_{S2}$, $Q_{opt} = 1$ for the row. Otherwise, $Q_{opt} = 2$ for the row. This gives Equation (5.7). This completes the proof. □

5.4.3 Determination of Optimum Q by using Simulations

As seen in the previous section, it is complex to decide the optimum value of Q, Q_{opt}, mathematically. Therefore, to take a decision on Q_{opt}, a simulation technique is also proposed for scenarios which have not yet been addressed mathematically. This technique is more useful when the sizes of the data items are non-uniform or more than

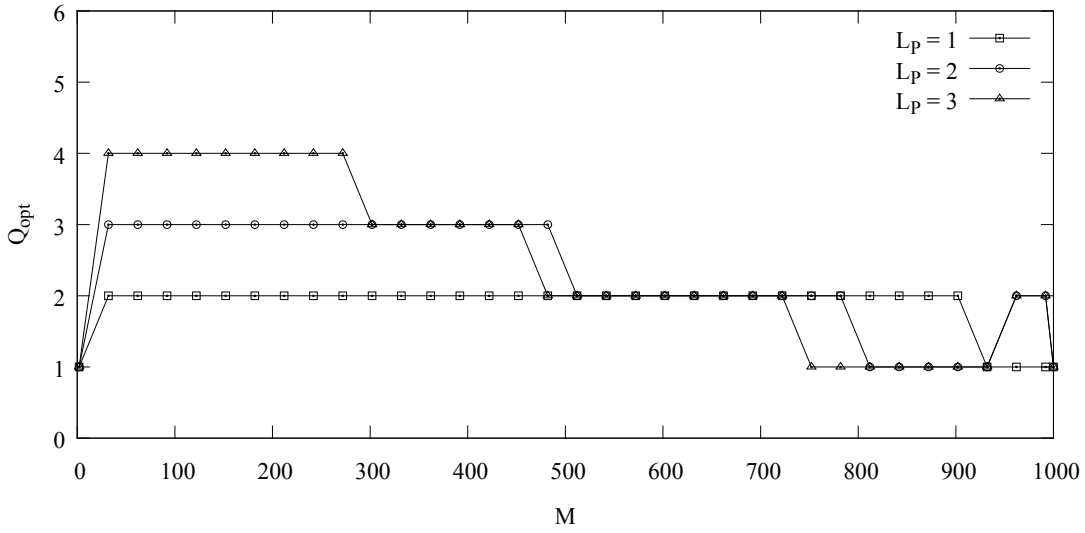
one parallel *loads* are allowed. The simulation mechanism will be described in more details in Chapter 7 together with our simulations tool development.

In brief, each row is simulated with Q values from 1 to Q_{max} according to Theorem 11. The sample *load* and comparison times are taken from a brief benchmark test prior to the execution, as will be discussed further in Section 6.1.2. Then Gaussian distribution [Stein, 1981] is used to extrapolate the data and the simulations are run on the extrapolated data. The simulations are sufficiently fast so that the influence on the total runtime is negligible.

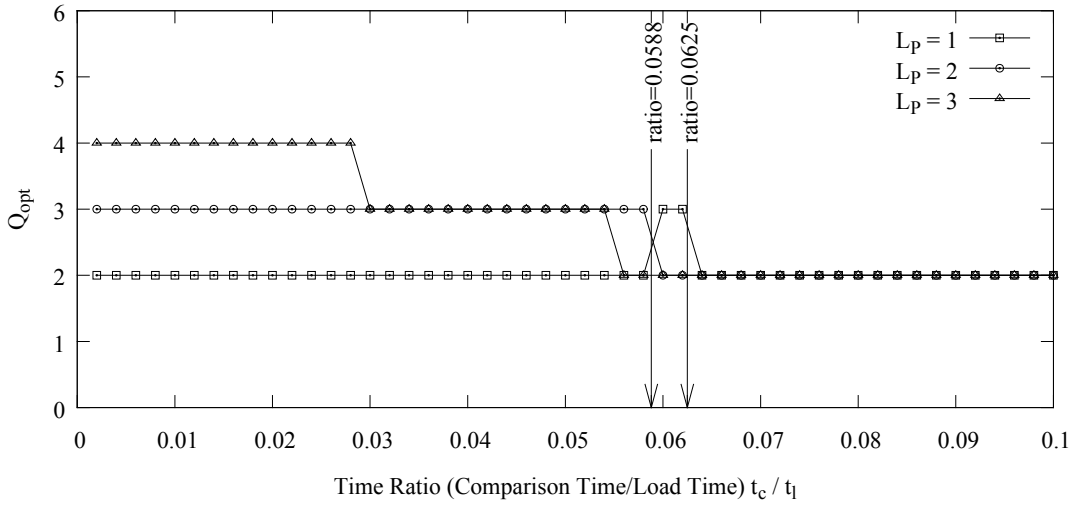
5.4.4 Validating Theoretical Results

In this section, experiments are used to validate the theories developed in Section 5.4.1 and 5.4.2 for mathematically calculating optimum Q. Simulations are used for the experiments in this section. A detailed discussion of the simulator tool that is used for the experiments is presented in Section 7.1.

According to Theorems 11 and 12, the optimum value of Q, Q_{opt}, is in the range $1 \leq Q_{opt} \leq 2L_P$ where L_P is the maximum number of efficient parallel loads. Figure 5.12 (a) shows that Q_{opt} always stays within the range $1 \leq Q_{opt} \leq 2L_P$ for all M values for each L_P in the experiment. Figure 5.12 (b) shows the same in a critical period of $\frac{t_c}{t_l}$ ratio. In Figure 5.12 (b), Q_{opt} stays within the range $1 \leq Q_{opt} \leq 2L_P$ when the conditions of the theorem are met (Equation (5.1) or Equation (5.2)). When deciding Q_{opt} with simulations, the row is simulated with all possible Q values, and the Q value that produces the highest rate of completing comparisons for the row is selected as Q_{opt}. The rate of completing comparisons is calculated by $\frac{\text{number of comparisons completed in the row}}{\text{time spent for completing the row}}$. It is seen from Figure 5.12, that Theorems 11 and 12 stands in the experiments.



(a) $\frac{t_c}{t_l} = 0.02$

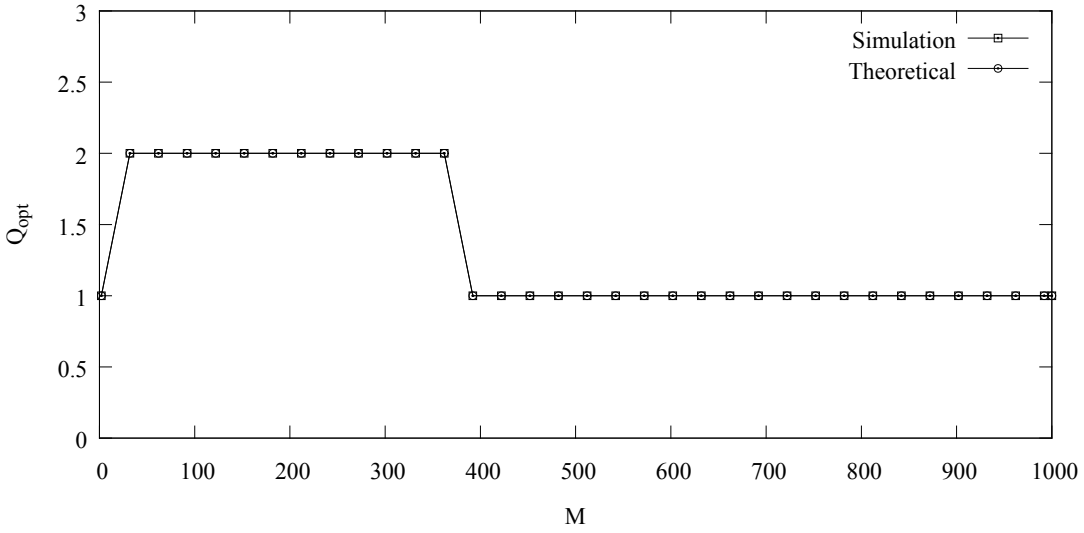


(b) $M = 250$; In between $\frac{t_c}{t_l} = 0.0588$ and $\frac{t_c}{t_l} = 0.0625$ conditions of the theorem do not meet

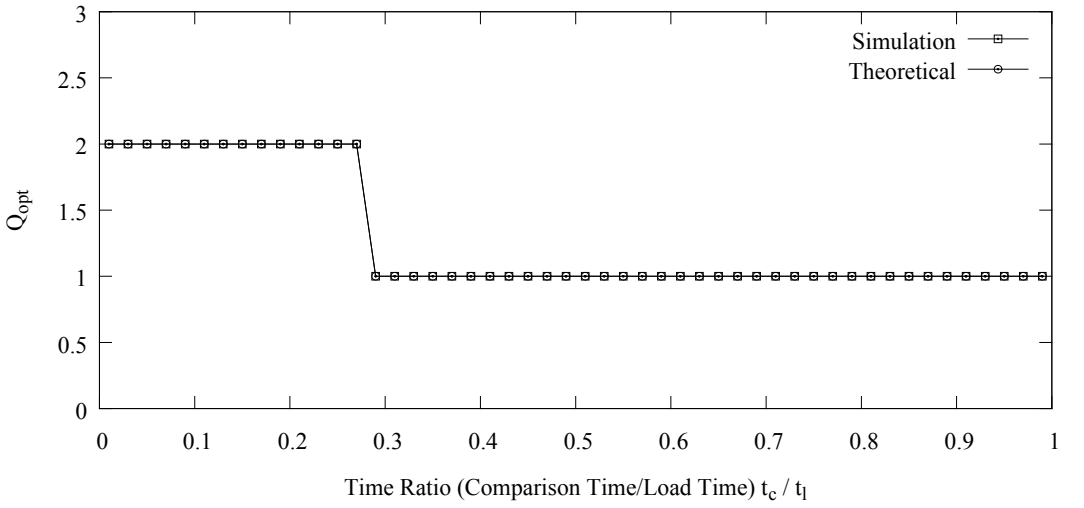
Figure 5.12: Value of optimum Q, Q_{opt} for the first row of Algorithm 7 (a) against the memory capacity, M (b) against $\frac{t_c}{t_l}$ ratio for different L_P values. Settings are $N = 1000, P = 16$, for (a) $\frac{t_c}{t_l} = 0.02$ and for (b) $M = 250$.

Theorems 13 and 14 estimate the value of Q_{opt} mathematically by using some approximations. Figure 5.13 shows that the estimated values using the theorems are similar to the values produced by simulations. Therefore, Theorems 13 and 14 stands in the

experiments.



(a) $\frac{t_c}{t_l} = 0.1$



(b) $M = 250$

Figure 5.13: Value of optimum Q, Q_{opt} for the first row of Algorithm 7 (a) against the memory capacity, M (b) against $\frac{t_c}{t_l}$. Settings are $N = 1000, P = 8, L_P = 1$, for (a) $\frac{t_c}{t_l} = 0.1$ and for (b) $M = 250$.

5.5 Summary of the Chapter

This chapter progressively developed a novel parallelization technique in the process of developing a new algorithm (Algorithm 7) for parallel execution of memory-constrained DIAC in shared-memory multi-core computers. The novel parallelization technique combines static and dynamic scheduling, and the scheduling is guided by a specific pattern developed based on the prior knowledge of the problem. The new parallelization technique was developed based on the strategies to overcome the problems in existing parallelization techniques to parallelize DIAC.

As we have seen in the case studies so far, the new algorithm (Algorithm 7) designed based on our parallelization technique has the potential of handling any DIAC problem efficiently. We will see in the next chapter (Chapter 6) that the extended version of Algorithm 7 is empirically shown to achieve near optimum efficiency in the experiments.

In addition, the optimum parameters for the proposed algorithm were theoretically derived for the most frequent scenarios. The theorems were experimentally verified. For the scenarios which are not addressed by the theoretical results, a technique to determine the optimum parameter settings by fast simulations was also proposed.

Chapter 6

Algorithm Implementation and Experiments

In the process of practical implementation of Algorithm 7 (in Chapter 5), there are a few problems to address, such as non-uniform data item sizes and unpredictable task runtimes. Therefore, this chapter proposes an extended version of the Algorithm 7 to address the practical implementation issues. A section of special techniques to improve implementation of the algorithm are also discussed. Followed by the implementation details, an algorithm designed specifically to parallelize preprocessing stage of the CV method in bioinformatics [Yu et al., 2010a] is also presented.

This chapter also experimentally verifies the performance of the proposed algorithm for parallelization of DIAC. The experiments are conducted on state-of-the-art shared-memory computer systems and by simulations.

6.1 Extending Algorithm 7 to Solve Practical Problems

This Section discusses the practical problems that occur while implementing the proposed Algorithm 7 in real world applications. Firstly, handling non-uniform sizes of data items is addressed and secondly, handling unpredictable *load* and comparison times is addressed. This section extends Algorithm 7 to Algorithm 8 which is capable of handling the practical issues in runtime.

6.1.1 Handling Non-uniform Sizes and Variable Q

As we already mentioned in Chapter 3, the maximum number of data items which can be held in memory, M, is not a constant. In a practical implementation of Algorithm 7, this issue must be considered for efficient computation of the DIAC. In addition, we have seen that the optimum size of Q, Q_{opt} is variable for each row. Therefore, this section extends Algorithm 7 to address these issues.

The extended Algorithm 7 is depicted in Algorithm 8. In this algorithm, the procedure INITIALIZE (N) is called before creating the threads. The new procedure, INITIALIZE (N) sorts the data items in descending order by the sizes of preprocessed data items. The loop at line 54 calculates height of each row. Within the loop, first it decides the optimum Q value, Q_{opt} using one of the methods discussed in Section 5.4. Then the maximum number of data items which can be loaded to the memory, M (from the next data items in the sorted list) is calculated for the row. The height of the row is calculated by deducting its Q_{opt} from its M. Then it records the heights in an array to be used at line 20. The value of Q_{opt} is not recorded. The reason is that once the row height is known which is the size of set U for the row, sufficient memory to load at least Q_{opt} data items for set V (at line 39) will be always spared. After finishing INITIALIZE (N), similar to Algorithm 7, each thread executes the procedure

COMPLETETASKS (N).

Importantly, Q_{opt} is only the minimum value of Q for a row. The size of V_{im} can always increase if sufficient memory is present. There is a major advantage of this method when the *loaded* data items have non-uniform sizes. When V_{im} progresses through the row, *loaded* data items get smaller (since the data items are sorted in descending order by their size) allowing more of them to be loaded to V_{im}. This can increase the performance by allowing more work to reside in memory. The usage of dynamic V_{im} size, also allows to choose lower Q_{opt} for the row as it is increasing towards the end of the row. Assume a situation is as following; When $Q = 1$, there are remaining set U comparisons to fill the gaps until the row is half completed. Then, due to the smaller sized data items towards the end of the row, V_{im} increases to two or more. In this situation, the algorithm takes advantage of dynamic sized V_{im} by selecting a smaller Q value and higher row height (fewer repeated loads).

Calculating and storing row heights statically and storing them before the comparisons begin, reduces the complexity of the algorithm implementation. For the most part, prior knowledge of the height of the each row allows the algorithm to use counters instead of loops to decide completion of a row and status of a set V data item. More information on this is discussed later in Section 6.2.

Algorithm 8 sorts the data items by their preprocessed sizes. As we have already seen in Chapter 3, sorting data items can significantly reduce the total time spent on *loads* when the *loaded* data items are large. However, this decision is adapted to the parallel algorithm based on three factors other than the advantages we have already seen such as dynamic Q:

1. Fewer repetitions for longer jobs The idea is to *load* the data items which take longer to *load*, less number of times by finishing all comparisons with them first. Typically the data items with larger sizes take longer to load than the data items

Algorithm 8 Scalable Data Pipeline Algorithm (Decentralized algorithm for each thread): The extended Algorithm 7 to handle non-uniform sizes. The algorithm is run by each thread in the system. A *load*, comparison and unload completion awakes all waiting threads. N is the number of data items to be compared.

```
1: Set U and V to be empty
2: V_im ← ∅ // Intialize the sub set of currently loaded items from V to be empty
3: compsU ← ∅ // Intialize list of available comparisons within U to be empty
4: compsV ← ∅ // Intialize queue of available comparisons within V ∩ U to be empty
5: row_heights = new Array() // Pre-calculated row heights
6: INITIALIZE(N)
7: procedure COMPLETETASKS(N)
8:     row ← 0 // Initialize current row count to be zero
9:     row_start ← 0 // Intialize current row's start position to be zero
10:    row_height ← row_heights[row] // Retrieve the height of the current row
11:    loadingU ← true // If true set U is being loaded
12:    INITIALIZESETS(row_start, row_height, N)
13:    while uncompleted comparisons remaining do
14:        if max parallel loads limit reached then
15:            COMPAREORWAIT()
16:        else
17:            if all comparisons in the current row are complete then
18:                Unload all data items in memory
19:                Increment row by one // Go to next row
20:                row_height ← row_heights[row]
21:                loadingU ← true // Indicate to load the next set U
22:                INITIALIZESETS(row_start, row_height, N)
23:            end if
24:            if loadingU then
25:                item = next not loaded and not being loaded data item in U
26:                if item is the last data item in U then
27:                    loadingU ← false // Finish loading set U
28:                end if
29:                LOAD (item);
30:                Add comparisons between loaded data items in U and item to
   compsU
31:            else
32:                for all item in V_im do
33:                    if item has completed all comparisons with U then
34:                        UNLOAD (item)
35:                        Remove item from V_im
36:                    end if
37:                end for
```

```
38:                 item = Next never loaded data item in V for current row_start
39:                 if memory is sufficient to load item then  // Higher priority for loads
40:                     LOAD (item)
41:                     Add item to V_im
42:                     Add comparisons between item and U to compV queue
43:                 else
44:                     COMPAREORWAIT()
45:                 end if
46:             end if
47:         end if
48:     end while
49: end procedure
50: procedure INITIALIZE(N)
51:     Sort all G_i by their preprocessed size in descending order
52:     row_start = 0
53:     i = 0
54:     while row_start < N do
55:         Q = Calculate optimum Q for this row
56:         M = Maximum number of data items can be loaded from G_{row_start}
57:         row_heights[i] = M − Q
58:         row_start ← row_start + row_heights[i]
59:         i++
60:     end while
61: end procedure
62: procedure INITIALIZESETS(row_start, row_height, N)
63:     U ← {G_x : x ∈ ℤ, row_start ≤ x ≤ row_start + row_height}
64:     V ← {G_x : x ∈ ℤ, row_start + row_height < x < N}
65:     Set V_im, compsU, compsV to be empty
66: end procedure
67: procedure COMPAREORWAIT
68:     if compV is not empty then  // Higher priority for comparison from V
69:         comp = front most comparison in compsV queue
70:         Remove comp from compsV
71:         COMPLETE (comp)
72:     else if compU is not empty then
73:         comp =A comparison in compsU list
74:         Remove comp from compsU
75:         COMPLETE (comp)
76:     else
77:         Wait until notified by another thread on a system state change
78:     end if
79: end procedure
```

with smaller sizes [CVTree, 2011, Yu et al., 2010b]. Therefore, the data items are sorted in descending order of their sizes, so that the first few set U are filled with data items with longer *load* times and all comparisons with them finishes first.

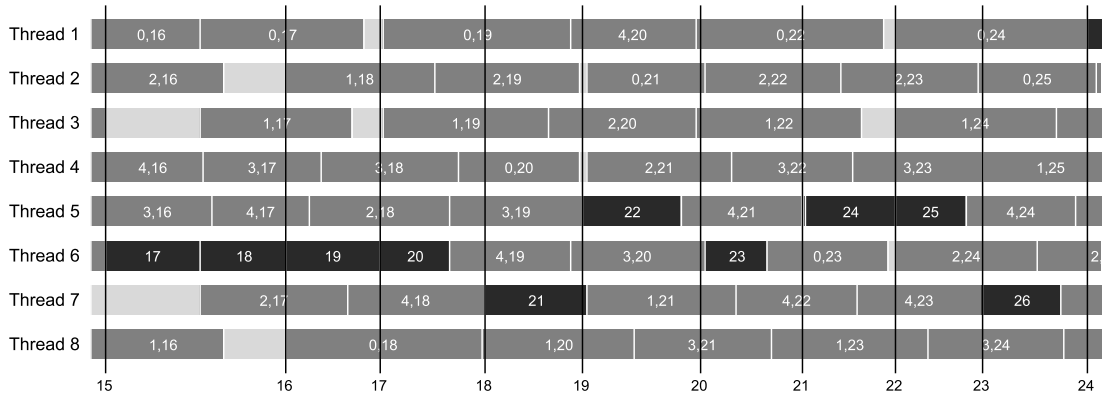
2. **More memory for set V** When large data items are at the end of a row, memory must be allocated for them to *load* throughout the calculation of the row. When those are completed first, the memory available for the rest of the calculation to *load* data items gradually increases.
3. **Complexity of Scheduling** When designing the algorithm, we took special care to reducing the computational complexity. When the data items are sorted in descending order, the sum of the first set in V_{im} queue is always the largest for the row. Therefore, once sufficient memory is spared for the first V_{im}, the algorithm does not have to manage memory for the rest of the data items to be loaded into V_{im}. This reduces the amount of calculations that the scheduling algorithm must perform for memory management.

6.1.2 Unpredictable Load and Comparison Times

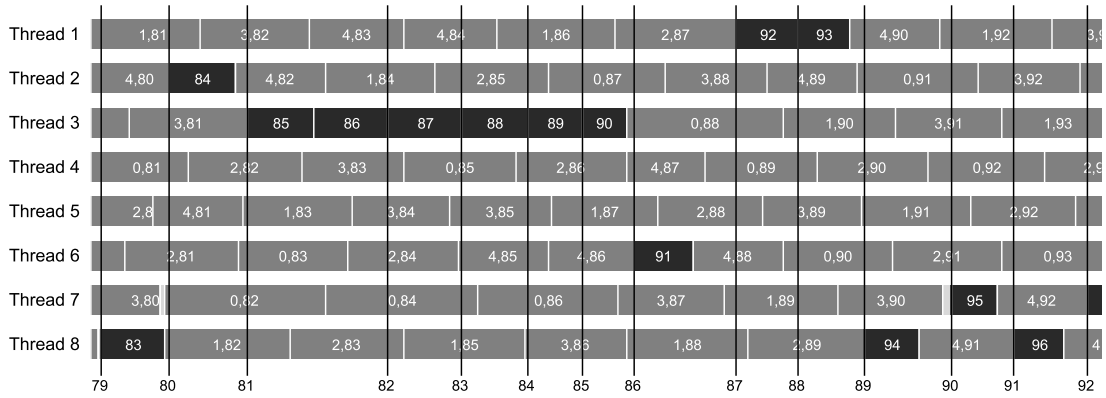
Handling unpredictable *load* and comparison times is important for the practical implementations of Algorithm 8. The runtime for loads and comparisons are unpredictable due to many factors which are beyond our control. Therefore, this section discusses how to overcome this challenge. Algorithm 8 is designed so that each thread has sufficient control over its own behaviour. As a result, the runtime behaviour of the algorithm is flexible allowing it to adjust for the variable runtime of comparisons and *loads*. For example, if one thread is taking longer than expected time to finish a job, other threads will try to manage with the available resources.

Figure 6.1 shows two snapshots of a thread diagram demonstrating the behaviour of

Algorithm 8. It has been drawn from actual data recorded in a log file. The thread diagrams show how the algorithm manages to withstand variation in the completion time of comparisons and *loads* without hindering performance. In Figure 6.1 (a), the threads are not receiving sufficient work to reach the system's full potential due to the bottleneck of *loads*. In Figure 6.1 (b), the system receives sufficient work to make all threads continuously busy. The diagrams also show that the threads are not waiting for any other thread unnecessarily.



(a) The system is unable to bring comparisons to memory at a sufficient rate to keep all threads busy



(b) The system is capable of bringing comparisons to memory at a sufficient rate to keep all threads busy

Figure 6.1: Two snapshots of the thread diagram for running Algorithm 7 in a computer with 8 cores.

Even though the runtimes are unpredictable, to choose the best Q value, we have to use estimated runtime for comparisons and *loads*. When the total runtime for a dataset

seems to be significantly long (say more than half a day), our algorithms will spend more time on benchmarking, so that more accurate simulations can be run to decide better Q values for each row.

In the applications which are used for the case studies and examples [CVTree, 2011, Yu et al., 2010a] in this thesis, the *load* time of a data item is strongly associated with its size. Therefore, the sample runtimes are taken after sorting the data items by the size or, if available, by the size after preprocessing (if the preprocessed data items have been written to the disks). The samples are chosen at the same intervals from the sorted list, and *load* times are measured for the samples. Then the non-measured *load* times in the dataset is extrapolated using the standard deviation and mean of the measured samples. Gaussian distribution [Stein, 1981] is used to extrapolate the data. For the comparison times, the mean of the measured comparison times is used since the variance of comparison times is small in the example applications [CVTree, 2011, Yu et al., 2010a].

6.2 Algorithm Implementation

While implementing Algorithm 8, a 'read-write' lock must be used to avoid race conditions between the threads while taking decisions. The lock should only allow a thread to enter into the decision making segments in the routine COMPLETETASKS(N). The lock should be released before starting a *load* or comparison within the routine. After finishing the work, the thread should acquire the lock again to modify the system status. As the unloads takes negligible time to complete, they can be completed while holding the lock. More details about the usage of the locks is discussed in Section 7.2 while describing our framework used for the implementation.

In Algorithm 8, there are two key decisions frequently made by each thread. The

decision process can be computationally intensive if not implemented properly. The first decision is whether or not all comparisons in the row have finished (at line 17). If a global task queue is used, all remaining tasks must be searched to take this decision. Because of the prior awareness of the row height, our approach is to use a 'counter' to assist the decision. When a new row is initialized within the condition at line 17, the counter is set to zero, and the number of comparisons in the row is calculated and stored. When a thread completes a comparison, the counter is incremented by one. Once the counter reaches the number of comparisons in the row, the threads have finished all comparisons in the row. We use a similar mechanism with a counter to decide whether all comparisons have competed or not at line 13.

The second decision is that before unloading a data item in set V (V_{im}) from memory, it must be certain that all comparisons with the item are completed (at line 33). Instead of searching through a remaining tasks queue for each data item to decide whether all comparisons with the data item is finished or not, we use an array of counters for this. An array of counters, each of which corresponds to a data item in the dataset, is initialized first. Upon completion of the *load* of a data item, the value of the corresponding counter is initialized to the number of comparisons related to the data item (equal to the height of the row), if it is in set V. Then each time a comparison related to the data item is completed, the corresponding counter is decremented by one. Therefore, when the corresponding counter of a data item in V_{im} is zero, all comparisons with the data item have been completed and the data item can be unloaded.

Because of the optimizations and the lower complexity of the scheduling algorithm, the time spent on scheduling is significantly low as we will see in the experimental results. In Figure 6.1, it cannot be seen that a thread is delayed for decision making process. It is not visible due the very small value compared to others.

6.3 Preprocessing Data Items in Bioinformatics CV method

This section focuses only on the applications that we used for case studies and examples [CVTree, 2011, Yu et al., 2010a]. These applications are for the CV method [Yu et al., 2010a] calculations on a set of genomic sequences (GSes) as discussed in Section 2.1.3. In the *load* process, *composition vectors* are generated based on the genomic sequences. The *composition vectors* are then pair-wise compared to a DIAC to generate the output correlation matrix.

Loading pre-written *composition vectors* from the disk is faster in many systems than preprocessing them from scratch each time when it is loaded. One of the challenges of the preprocessing phase of the CV method [Yu et al., 2010a] is that the required memory cannot be estimated prior to calculating the *composition vector*. However, it is important to manage the memory also in this phase within the physical memory limitations. Using memory over the available physical memory will encourage the usage of virtual memory. As we have seen in the experiments in Section 3.5.4, involvement of virtual memory can significantly hinder the execution speed. This is more challenging specially when *composition vectors* are calculated in parallel. This is because when more threads are performing preprocesses, more memory is required and the chance of exceeding the available memory increases. We assume that every system has sufficient available memory to preprocess at least one genomic sequence (GS) at a time.

This section proposes a parallel algorithm to overcome the challenges in parallel preprocessing. Typically, longer GSes require more memory in the preprocessing stage. Therefore, we use a heuristic that processes data items requiring less memory first and keeping more threads busy initially. Then the data items that require more memory are processed at the end, by a smaller number of threads if memory is limited. This decentralized algorithm is presented in Algorithm 9.

Algorithm 9 Parallel Algorithm to preprocess data items (Decentralized algorithm for each thread). Each thread in the system executes this algorithm. N - number of data items to be preprocessed.

```
1: procedure PREPROCESSALL(N)
2:     Sort data items by their length
3:     continue = true
4:     while continue and data item are remaining do
5:         G_i = next data item in 'to be preprocessed' list
6:         Remove G_i from 'to be preprocessed' list
7:         continue = PREPROCESS (N)
8:         if continue == false then
9:             Add G_i back to the 'to be preprocessed' list
10:            Reallocate memory to other threads
11:        else
12:            while limit for parallel disk writes is reached do
13:                Wait until notified
14:            end while
15:            Write G_i to the disk
16:            Reset used memory of the thread
17:            Notify all waiting threads
18:        end if
19:    end while
20: end procedure
21: procedure PREPROCESS(G_i)
22:    if REQUESTMEMORY (Memory required next) == false then  // If memory
    required is not granted
23:        Release memory acquired so far
24:        return false
25:    end if
26:    ...
27:    if REQUESTMEMORY (Memory required next) == false then
28:        Release memory acquired so far
29:        return false
30:    end if
31:    ...
32:    return true
33: end procedure
```

Continued in the next page...

```
34: procedure REQUESTMEMORY(required)
35:     if the allocated memory for the thread is insufficient for required memory then
36:         Increase used memory of the thread by required
37:         return true
38:     else
39:         return false
40:     end if
41: end procedure
```

Before the process begins, the GSes are sorted in ascending order by their size at line 2. Initially the total available memory is divided equally and allocated to each thread. The idea is to make more threads busy initially, since front most GS in the sorted list require less memory. The loop at line 4 continues until the thread terminates due to insufficient memory or the application completes preprocessing all GSes. The variable $continue$ will get $false$ if the memory allocated for the thread is not sufficient.

As seen in the procedure PREPROCESS(G_i), each time the process goes to require memory it has to be checked against the allocated memory for the thread (line 22 and 27). This check is done in large chunks of memory (i.e. infrequently), so that the overhead created is insignificant. If the memory allocated to the thread is insufficient to allow the required memory (i.e. procedure REQUESTMEMORY returns $false$), the process pre-empts and releases any memory it is holding. Then the failure is propagated to the main procedure (PREPROCESSALL(N)). The procedure REQUESTMEMORY($required$) remembers previously successful memory requests and checks the available allocated memory for the thread against the requested memory.

If preprocessing procedure fails due to low memory (i.e. procedure PREPROCESS returns $false$), the thread will terminate and the memory allocated to the terminating thread is equally reallocated to the remaining threads. The procedure REQUESTMEMORY($required$) uses locks so that if a thread is terminating, other threads wait until the termination and the memory re-allocation finishes before pre-

empting another preprocessing thread on insufficient memory. This ensures that the threads does not terminate when sufficient memory is going to be allocated soon.

If the preprocessing is successful the thread will wait until it gets access to the disk. Usually the disks are capable of sequential writes. As a result, too many parallel write requests can hinder their performance. Therefore, the main procedure wait until any other thread releases the disk and notifies it. Once, it get access to the disk, the preprocessed GS is written to the disk and all threads waiting for the disk are notified.

As will be seen later in the experimental results, this algorithm can increase the preprocessing rate reasonably, given that sufficient memory is present for all cores to work simultaneously. When memory is limited, especially towards the end of the preprocessing phase, threads starts terminating due to low memory. However, due to initial parallel execution of calculations, the algorithm still manages a good performance gain in limited memory conditions, even when the disks are only capable of sequential reads. The most important quality of the algorithm is that the application never exceeds a specified memory limit. Also the memory limit can be increased or decreased at runtime while threads are still working. To reduce the memory limit, one or more threads can be terminated by setting their allocation to zero. After reducing, if memory limit increases later, more threads can be created.

6.4 Experimental Results

The experiments to be carried out have two objectives:

- investigating the performance and behaviour of the proposed algorithm (Algorithm 8) in various conditions; and
- demonstrating the performance of the Algorithm 8 in real computing platforms.

We use three different computers with a memory (RAM) of 4 GB, 64 GB and 256 GB and cores 2, 8 to 16. The Intel® Hyper Threading™ technology is disabled on these computers for more accurate performance evaluation. To emulate a system with less memory such as 8 GB and 24 GB, the experiments are conducted in the same computers with a pseudo memory limit.

6.4.1 Behaviour of Algorithm 8

Most of the experiments in real computing platforms take very long time to complete (days or weeks). In addition, access to some system resources such as disks capable of parallel disk reads is limited. Therefore, to analyse the behaviour of the algorithm, simulations are used. The analysis is based on the four factors which affect the behaviour of the algorithm and are discussed in Section 5.3.2.

The speed-up, S is calculated using Equation (6.1) where T_p is the runtime of the parallel algorithm (Algorithm 8) and T_s is the runtime of the sequential algorithm (Algorithm 5).

$$S = \frac{T_s}{T_p} \tag{6.1}$$

Since we demonstrated that Algorithm 5 is faster than existing sequential memory management algorithms for DIAC, we use it as the reference for calculating the parallel algorithm's speed-up. In each experiment, both algorithms use the same system and dataset properties. Algorithm 8 uses the optimum Q value for each row based on the theories developed in Section 5.4. It should be noted that the sizes of the data items, t_c and t_l, are uniform for the simulations in this section. The simulator tool is discussed in more details in Section 7.1.

The ratio between *load* time and comparison time

The ratio between *load* time (t_l) and comparison time (t_c) is one of the major factors effecting the parallel gain of the algorithm. Therefore, we analyse the speed-up of the parallel algorithm against the t_c/t_l ratio, while keeping other factors unchanged.

To examine the effect of maximum parallel load limit, L_P, we have drawn three graphs with L_P ranging from 1 to 3. As seen in Figure 6.2, when the ratio t_c/t_l increases the system approaches near optimum efficiency. The reason is that when the comparison times dominate, more work is available for the threads, even though the memory is limited. Therefore, the effect of the memory and I/O bottleneck gradually diminishes when the amount of work a single *load* brings increases. When *load* time dominates, threads do not get sufficient work since the memory is limited (i.e. available comparisons are limited) and the *loads* become the bottleneck.

When L_P increases, the bottleneck of *loads* gradually diminish. As a result the rate of bringing comparisons to memory increases. Therefore, the system approaches near optimum efficiency at a lower t_c/t_l ratio at higher L_P values compared to the lower L_P values.

The effect of memory capacity

Similar to Algorithm 5 for uni-processor platforms, Algorithm 8 is designed to take the full advantage of the available memory. However, Algorithm 8 benefits more from memory than Algorithm 5 since Algorithm 8 can execute comparisons in parallel. Figure 6.3 shows the speed-up (S) of Algorithm 8 versus the available memory (maximum number of data items memory is capable of holding, M). It can be seen in Figure 6.3, when more memory is available threads are working more on comparisons and increases the speed-up due to the increasing number of comparisons available in mem-

ory. Similar to Figure 6.2, the rate of increasing the speed-up rises when L_P increases because the throughput of *loads* increases when L_P increases. After a certain memory size, the rapid increase in speed-up stops and starts to gradually approach near optimum speed-up. The point where the rate of increasing speed-up with M slows down is when all threads becomes continuously busy throughout the row after loading set U.

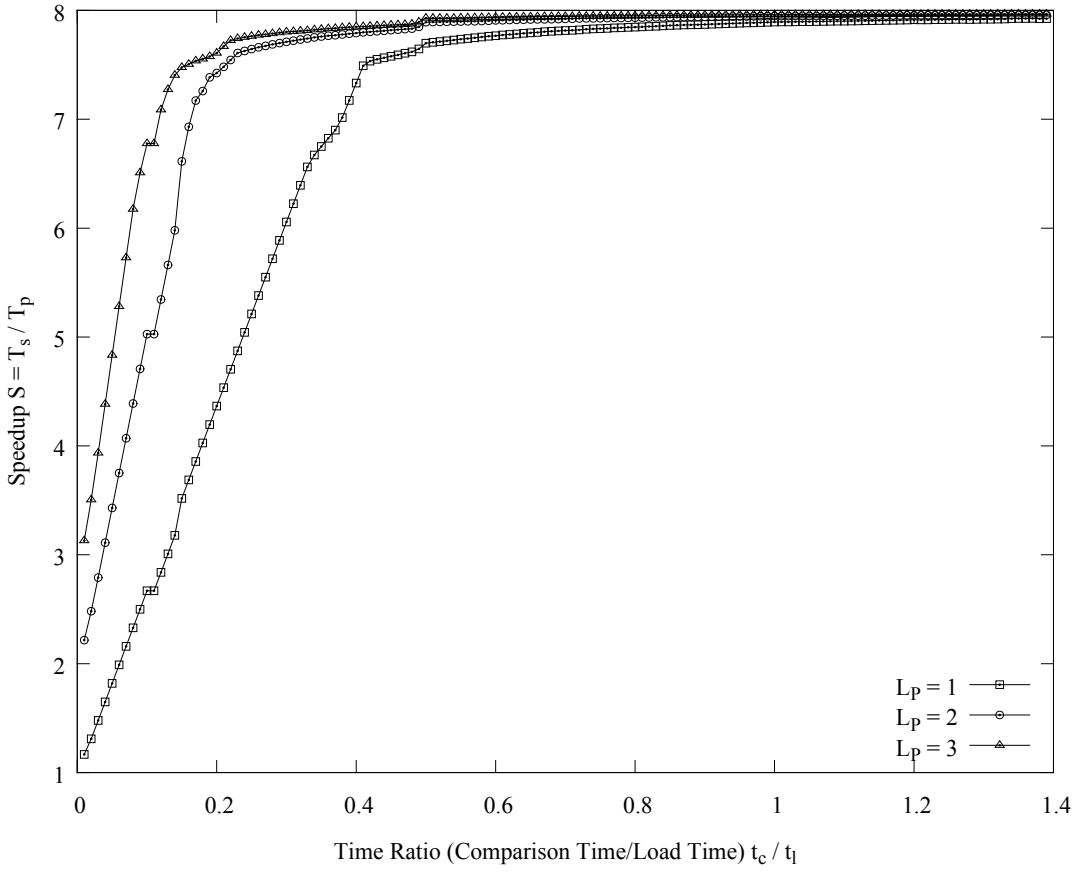


Figure 6.2: Speed-up of Algorithm 8 against t_c/t_l ratio for different L_P values. Settings are $N = 350, M = 15$ and $P = 8$.

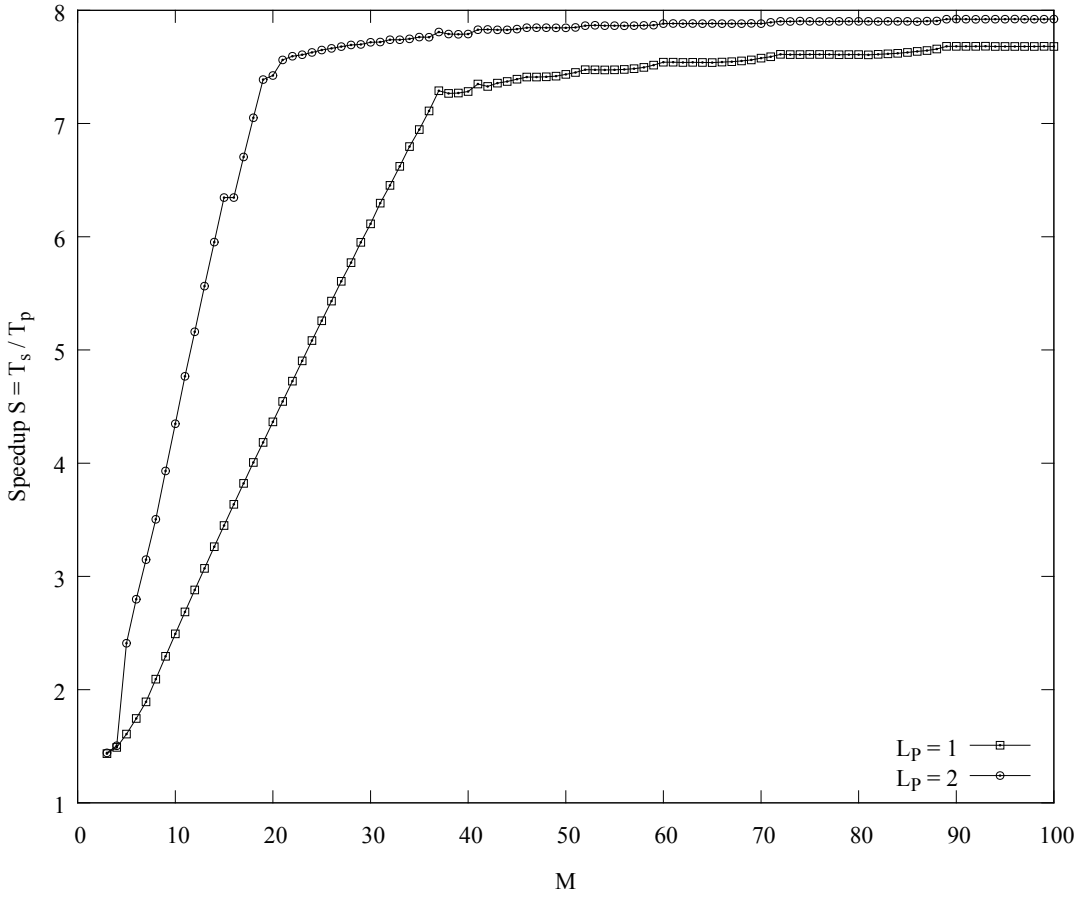


Figure 6.3: Speed-up of Algorithm 8 against the memory capacity (i.e. the maximum number of data items which can be held in memory, M) for different L_P values. Settings are $N = 350, P = 8, \frac{t_c}{t_l} = 0.2$.

Number of threads

According to Amdahl's law [Amdahl, 2013], generally the speed-up of a parallel implementation should increase when the number of processors increases. Also, the rate of increasing the speed-up should decrease when the number of processors increases. However, in memory-constrained DIAC the rate of bringing comparisons into memory is limited by the maximum throughput of *loads* and the tasks (i.e. *loads* and comparisons) cannot be further divided into parallelly executable fractions. Therefore, as

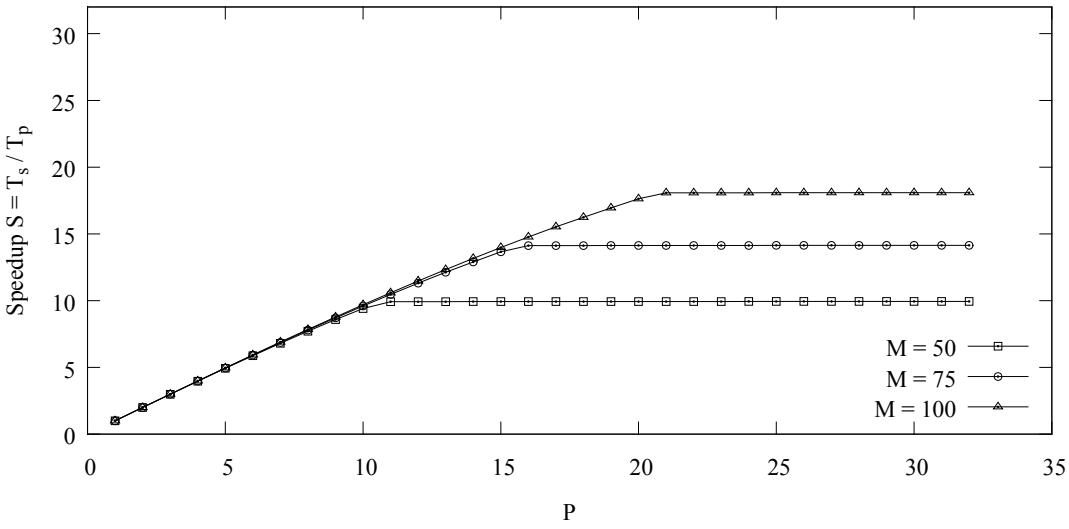
seen in Figure 6.4, the speed-up does not increase after a certain limit even though the number of processors increases. This limit occur when the system has saturated the maximum rate of bringing comparisons to the memory. At this saturated state, the *loads* are carried out continuously in each parallel channel of loads. Understandably, when more memory is available, the point of saturation of speed-up increases.

6.4.2 Performance of the Proposed Algorithm 8

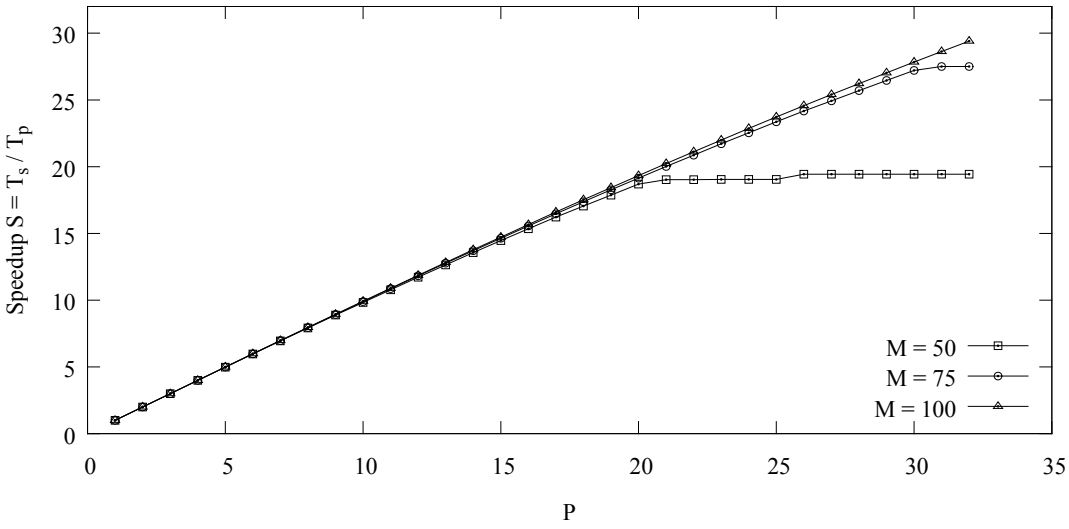
In this section experiments are conducted using the real implementation of the proposed algorithm in real computing platforms. Algorithm 8 is implemented in C++ programming language and GCC compiler with optimization level three is used. As we discussed in Section 3.5, we improved the original application written by Yu et al. [2010a]. For these experiments we use a version (in both sequential and parallel programs) with further improved internal memory management by implementing Wang's proposals. The optimizations proposed by Wang [2009] significantly reduces the memory required for preprocessing genomic sequences (i.e. generation of *composition vectors*).

Every implementation in this section uses the method of writing the *composition vectors* to the disk and reading them from the disk later. As seen from the profiling data in Table 6.1 acquired from the two different computer set-ups where the experiments are run, it can be clearly seen that reading a previously generated *composition vector* from the disk is much faster than generating it repeatedly. As our memory management algorithm requires accurate sizes of the *composition vectors* prior to the comparisons begin (to avoid over using memory), the *composition vectors* are written to the disk when each *composition vector* is generated for measuring the size.

We used two different powerful computers from two of the high performance computing laboratories. One is from the laboratory called 'Big Data' and other one is from



(a) $L_P = 1$



(b) $L_P = 2$

Figure 6.4: Speed-up of Algorithm 8 against the number of processors with different L_P and M. Settings are $N = 700$, $\frac{t_c}{t_l} = 0.2$ (a) $L_P = 1$ and (b) $L_P = 2$.

the 'QUT HPC' [QUT, 2014] laboratory. We call them 'Big Data' and 'HPC' here onwards. We also ran the experiments on a typical desktop computer with strictly limited memory which is called 'Desktop' here onwards. Table 6.2 lists the properties of these computers.

Table 6.1: Comparison between average time spent for generating a *composition vector* from scratch and time spent for reading it from the disk

Data Set	Computer	Generation (sec.)	From Disk (sec.)
Data Set 1	HPC	163.613	24.623
	Big Data	180.339	6.451
Data Set 2	HPC	42.094	0.015
	Big Data	23.415	0.009

Table 6.2: The properties of the computers which are used for the experiments

Property	**Big Data**	**HPC**	**Desktop**
Operating System:	Red Hat Enterprise Linux Workstation, release 6.4	SUSE Linux, version 3.0	Ubuntu version 11.04
Processor:	Intel® Xeon® Processor E5-2609	Intel® Xeon® Processor E5-2670	Intel® Core™2 Duo Processor E8400
Number of Cores:	8	16	2
Single Core Speed:	2.40 GHz	2.6 GHz	3.00 GHz
Processor Count:	2	2	1
Cache:	10 MB	20 MB	6 MB
RAM:	256 GB	64 GB	4 GB
Hyper Threading:	Disabled	Disabled	Disabled

Data Sets

Our experiments are based on the same data sets that we described in Section 3.5 (Data Set 1 and Data Set 2). However, we generated a few bigger data sets based on Data Set 1 and Data Set 2. It was done by randomly combining sub-genomic sequences from base data set's input files into new input files. Since every sub sequence in the input files of a data set represent a genomic sequence of a species, the generated data sets represent real data sets in large scale. In the data generation process, the mean, standard deviation, minimum and maximum of input file sizes in all generated data sets are kept similar to the base data set by using Gaussian distribution [Stein, 1981].

Table 6.3 lists all data sets that are used for experiments with a name to be used for later references.

Table 6.3: The list of data sets used in the experiments and their properties

Base Data Set	Name	Number of Files	Aprox. Size in Memory (GB)
Data Set 1	Set 1x	900	300
Data Set 2	Set 2xs	700	20
Data Set 2	Set 2x	2000	56
Data Set 2	Set 2xl	11000	300

Experiments

Since we experimentally validated in Chapter 3 that our single-core memory management algorithm is better than existing algorithms for DIAC, we compare the speed-up of our memory management algorithm compared to our single core algorithm (Algorithm 5). We are not running experiments with the original algorithm used by Yu et al. [2010a] which only hold two genomic sequences in memory, due to extensively long runtime with large data sets.

Table 6.4 shows the speed-up in various set-ups. The results are visualized in Figure 6.5. The maximum memory of the HPC computer is 64 GB. However to avoid virtual memory we used a memory limit of 57.6 GB (i.e. 90% of 64 GB) according the experiments conducted earlier. For other experiments we used a pseudo memory limit for demonstration purpose in low memory situations.

Table 6.4 shows that our algorithm has achieved a good speed-up with both data sets in Big Data computers. 7.81 and 7.86 speed-ups have been achieved by Algorithm 8 in a 8 core system. The combined speed-up is slightly less because of the lower parallel gain in preprocessing by Algorithm 9. The main reason for the lower parallel gain while

Table 6.4: A comparison between our sequential memory management algorithm (Algorithm 5) and our parallel algorithm (Algorithm 8).

Data Set	*Computer*	*Memory Used (GB)*	*Number of Threads*	*Algorithm 5 Total (hours)*	*Algorithm 8 Total (hours)*	*preprocessing Speed-up*	*Comparisons Speed-up*	*Combined Speed-up*
Set 1x	Big Data	57.6	8	33.64	4.85	1.81	7.81	6.94
Set 1x	HPC	57.6	16	87.34	21.39	14.89	3.77	4.08
Set 2xl	Big Data	24	8	327.42	43.31	4.30	7.86	7.80
Set 2x	HPC	8	16	13.45	0.91	7.40	15.77	14.78
Set 2xs	Desktop	3.12	2	1.62	0.97	1.26	1.72	1.67

preprocessing is the slower speed for data writing to the disk than reading speed. In the 16 core system in HPC with Data Set 2x, Algorithm 8 has achieved 15.77 speed-up.

As seen in the results, Algorithm 8 with Data Set 1x has achieved 7.81 speed-up in the Big Data computer. Even with the slower disk throughput in HPC (slower *loads*), Algorithm 8 achieved a speed-up of 3.77 with the same data set and the same memory limit. Achieving a closer to the number of cores speed-up shows the efficiency of our algorithm. It also shows the lower scheduling overhead created by the scheduling algorithm as a result of the lower complexity of the scheduling algorithm.

6.5 Summary of the Chapter

This chapter developed Algorithm 8 which addresses the practical implementation issues by extending Algorithm 7. Algorithm 8 achieved near optimum efficiency as seen

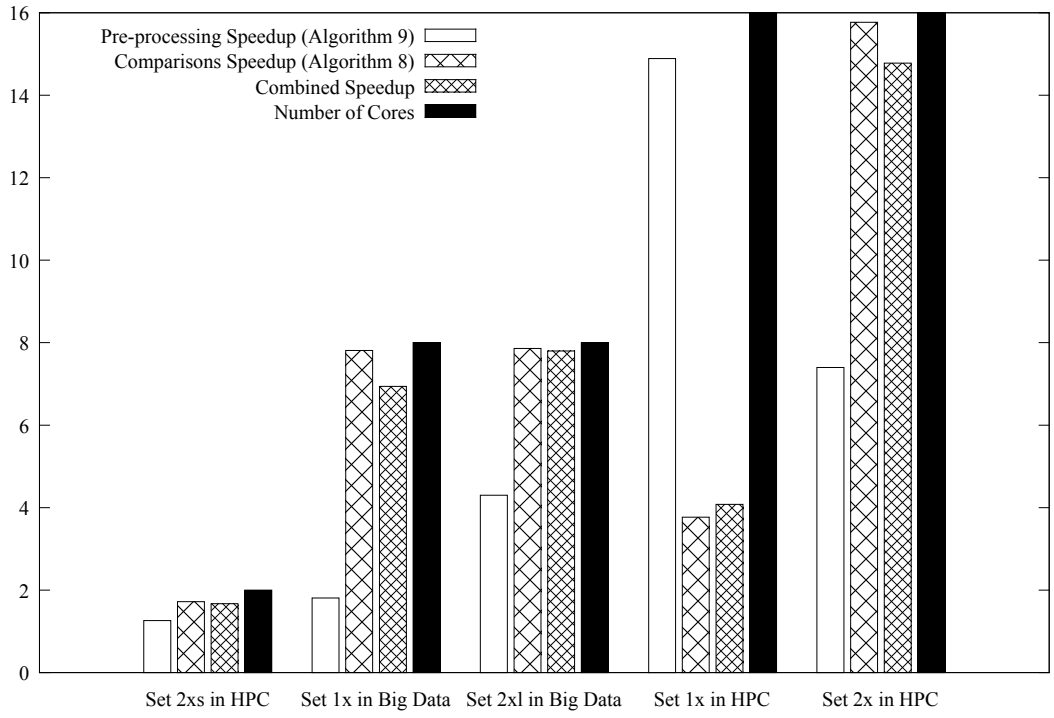


Figure 6.5: A visualization of the performance of Algorithm 8 based on the results shown in Table 6.4.

in the experiments. The algorithm significantly increased the speed of existing DIAC applications. For instance, our algorithm reduced the runtime of the application by Yu et al. [2010a] with a large data set to nearly 2 days from nearly 14 days (two weeks) with an impressive 7.86 speed-up in a 8-core shared-memory system. In another instance, our algorithm achieved a 15.77 speed-up in a 16-core shared-memory system reducing nearly 16 hours of runtime to nearly 1 hour.

The proposed algorithm scales well in different computer systems as seen in the experiments. It is scalable in the sense that it can efficiently manage memory while taking the advantage of the available computing resources, in any shared-memory multi-core system with different settings. The scalability of the algorithm and its behaviours under various settings were also verified by using simulations. Our algorithm's ability to

take advantage of disks capable of efficient parallel reads makes it scalable in advanced systems with such disk systems. Overall, the proposed algorithm minimizes the affect of limited memory on the parallel gain of DIAC applications by efficiently utilizing memory, disks and processors while strategically handling *loads*.

Chapter 7

Simulators, Visualizers and Frameworks

The cycle of analysing, developing strategies, experimenting and improving is the methodology followed by us to solve the complex DIAC parallelization problem. This chapter discusses a simulation environment and a parallelization framework for DIAC developed to aid the analytical and experimental phases of the research. This chapter answers the research question 3 specified in Section 1.3. The question is: how to evaluate the proposed algorithms, analyse their behaviour under various conditions (dataset and system properties) and validate theoretical results.

The chapter is organized as follows. First, the simulator tool which achieves an extremely fast simulation speed based on the specific contexts of our research is presented. Then the parallel framework is presented for the rapid modification and implementations of scheduling algorithms for DIAC parallelization, and which clearly separates the decision making logic from the implementation concerns. Then a unique visualization tool is presented which is useful for analysing parallel algorithms designed for DIAC.

The contributions of this chapter are as follows.

- A novel fast simulator tool is designed and developed to exploit the advantage of assumed uninterrupted and continuous processor-task association of the proposed parallel algorithms. The tool is lightweight and extremely fast to simulate long running applications, and also flexible to be utilized by other research in the similar context as our research.
- A novel parallelization framework specifically designed for DIAC to separate decision making logic from other system concerns such as thread-safety is designed and developed. The framework also allows dynamic adaptation of different scheduling strategies in the runtime.
- A unique visualization tool is also developed to analyse the behaviour of DIAC in runtime.

7.1 The Simulator

Simulators are important tools to analyse the characteristics of an algorithm and evaluate its behaviour. Faster execution of the simulations than the actual implementation of an application is a key advantage of using a simulator. In addition, the simulators are capable of emulating theoretical environments such as zero execution time for a code segment.

As discussed in Section 2.7, all of our expectations from a simulator cannot be fulfilled by using an existing simulation environment. Therefore, we design and develop a novel simulation environment to meet all our requirements. The flexible design of the new simulator allows easy modifications and utilization by other researches with similar requirements. The following is a list of expectations from a simulator in our research, which are addressed by the proposed simulator:

Speed Typically the bioinformatics programs that we use for case studies run for a very long time. Some data sets take days or weeks to complete. In addition, the simulations are used to estimate the optimum parameters for the proposed parallel algorithm as discussed Section 5.4. To minimize the overhead of scheduling, the simulations must be extremely fast.

Theoretical Behaviour The simulator should be able to theoretically simulate timing of some sections of the simulated program such as zero scheduling cost. This feature allows validations of theoretical results using the simulator.

Flexibility Ability to change the simulated program is an important feature in a simulator, so that various implementations can be transformed to the simulation conveniently.

Using External Timing The simulator should be able to accurately use user-provided timing for each task, so that the imagined and harvested task execution times can be used for the simulations for experiments.

Deterministic Simulation Multi-threaded programs usually do not behave in the same way twice and the bugs and other problems are difficult to reproduce. However, two simulations should behave exactly the same way if the inputs are the same.

Lightweight To use simulations for optimum parameter prediction in real runtime, it must be lightweight and should not use excessive amount of memory, since the scheduler's memory usage has to be tightly limited to achieve better memory management.

Estimations Estimating the total runtime of an application is important in many scenarios. For instance, High Performance Computing facilities expect users to provide a realistic runtime estimation for job scheduling purposes. Therefore, given the sample runtime of the tasks, the simulator must be able to predict the total runtime of an application.

Intergration The simulated annealing techniques discussed in Section 4.1.5 utilise the simulator environment to build random schedules. Therefore, the simulator

must have the ability to integrate with other applications.

To meet above listed requirement, a simulation environment is designed in the next section.

7.1.1 Simulator Design

The simulator is implemented as a sequential program without using threads. The debugging process is far simpler in a sequential program than debugging a multi-threaded version of the simulator. Moreover, sequential implementation better suits our design.

The simulator uses its own clock which controls the execution of the tasks in every thread. The basic time unit of the simulator is called a *tick* here onwards. A *tick* can represent any amount of time. Usually it represents 1 millisecond in actual time. The threads can schedule a task to finish at a certain *tick*. For example, if a comparison which has a runtime of 5 *ticks* starts at 100^{th} *tick* it will be scheduled to finish at the start of 105^{th} *tick*. The scheduler in the simulator ensures that the code after the comparison is executed at 105^{th} *tick*.

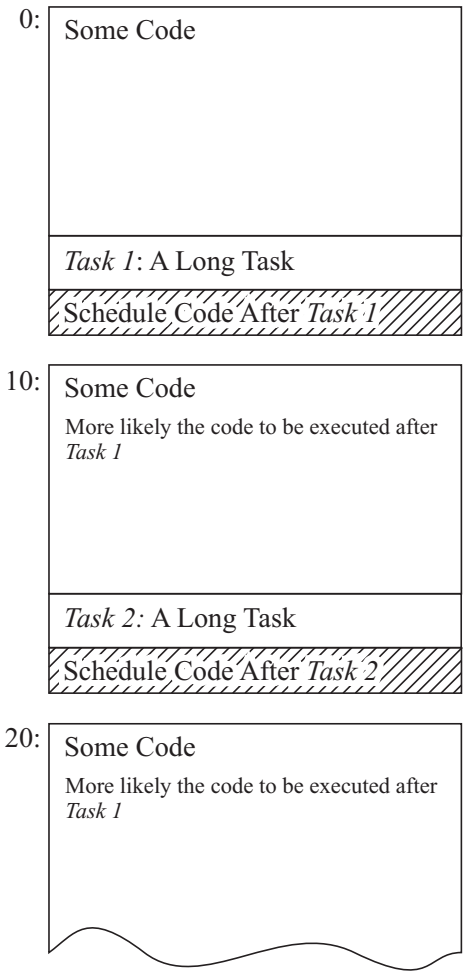
It should be noted that, to achieve extremely fast simulations of long running programs, we try to make the simulator as simple as possible. We take the advantage of two properties of the proposed algorithms to increase the speed of the simulations. Firstly, the tasks does not change the system status while it is being executed (only at the start and the end it changes the system status). Secondly, the algorithms assume that once a task is started by a thread, it is assigned to a single processor until it finishes. It is assumed that the tasks are not interleaved or interchanged between processors.

The assumption of that the system status changes occur only at the start and end of a task can be justified as follows. The long running tasks in the DIAC are the tasks *load*,

comparison and scheduling (if the time spent in scheduling process is accountable). Assume that a task starts at *tick* A and finishes on B. The task executes between time A and B (inclusive) in the real runtime. However, no change to the system is done by any of the above mentioned tasks in between A and B. At the end of a *load*, a new GS is bought into memory and at the end of a comparison, the comparison is removed from the system and the results are stored. Even though the unloads are performed within the scheduling process, the changes are only effective in system after it's completion, since locks only allow one thread to enter the decision making process. As a result, in the simulations the representative code for completing a task can be executed at finishing *tick* (B) without effecting the behaviour of the execution. This way a huge increase in speed in the simulation process can be achieved by skipping the CPU cycles in between the tasks and is discussed later in this section. Due to this improvement, an application which runs weeks in a real system, usually take less than 30 seconds in the simulator.

Figure 7.1 shows how a program to be simulated is implemented in our simulator. The figure shows an implementation of a sample code within a thread. The code is broken into blocks by long running tasks. At the end of each block a long running task is started. In the interval between two code blocks the execution of the long running task is emulated by skipping *ticks* equal to the length of the task. The finishing *tick* for the long running task is calculated by adding runtime of the long running task to the current *tick* of the simulator. The code after the task is scheduled at the calculated finishing *tick*. As stated in the previous paragraph, the finishing code of the task also should be included at the beginning of the code executed after the task. The simulator's scheduler is responsible for calling the scheduled code block at the finishing *tick*.

For a clearer picture, a sample implementation of the thread code is given in Figure 7.1. Each thread is an instance of a class called 'SimulatorThread'. Each thread instance has a method called 'run()' which is invoked by the scheduler. When the 'run()' method




```
public class Algo1Thread extends SimulatorThread {

    public void run(int label) {
        long time = 0;
        switch(label) {
            case 0:
                // Some code

                // Start Task 1
                time = getCompletionTimeForTask1();
                schedule(10, time);
                break;

            case 10:
                // Complete Task 1

                // Some code

                // Start Task 2
                time = getCompletionTimeForTask2();
                schedule(20, time);
                break;

            case 20:
                // Complete Task 2

                // Some code

                // If this is the last line
                // terminate the thread by calling
                // terminate(); or loop back to a
                // previous code block
                break;
        }
    }
}
```

Figure 7.1: A depiction of how simulator executes a code and simulate long running tasks in between.

is invoked by the scheduler, it passes an integer value which indicates the label of the code block that should be executed next. The code within the run method is separated into blocks in a 'switch' statement. Based on the label passed in, the 'switch' chooses the code to be executed next (i.e. the code within the relevant case statement). At the end of a code block, a call is made to 'schedule()' method to inform the scheduler that the next code block's label and the scheduling time. Loops which go across multiple code blocks can be formed by scheduling a previous code block (if necessary with the current *tick*, so that it runs immediately). The initial code block must always have the

label of '0', since the initial call to 'run()' method passes block '0'. All variables which must persist the value through code blocks are declared as class level variables, so that they retain their value in every call to 'run()' method.

The scheduler uses a clock skipping technique to increase the speed of a simulation. At the end of each *tick* the simulator examines all active threads and finds the start *tick* of the next soonest scheduled task. As mentioned earlier, since the tasks cannot be pre-empted there will be no execution in between the current *tick* and the next soonest scheduled task's start *tick*. Therefore, the scheduler jumps from the current *tick* to the next soonest scheduled task's start *tick*. This significantly increases the speed of the simulation, compared to evaluating the threads in each *tick*. Many existing schedulers are unable to exploit this speed-up due to their complex underlying architecture designed to support more complex simulations.

Figure 7.2 shows a part of the class hierarchy of the simulator which is responsible for simulating threads. Each thread in the simulator is a children of the 'SimulatorThread' class. The children can have its (thread's) code within the 'run()' method. An instance of the child class can register in an instance of the 'Scheduler' class (there is only one 'Scheduler' instance for a simulation). The corresponding 'Scheduler' instance is responsible for calling each registered thread's 'run()' method at the correct *tick*. Therefore it keeps a list of the registered threads. The 'Scheduler' instance accesses the next *tick* and the label stored in every registered child class instance of 'SimulatorThread' (i.e. thread). If a thread is terminated, a flag is set in the corresponding thread instance once the 'terminate()' method is called. The 'run()' method of a terminated thread will never be called again and the terminated thread is unregistered from the 'Scheduler'. Upon registration the caller can schedule the first code block, based on the current *tick* in the simulator.

This design gives the flexibility to start a thread at any given time in the simulation and execute custom code in the simulator by extending 'SimulatorThread' class. How-

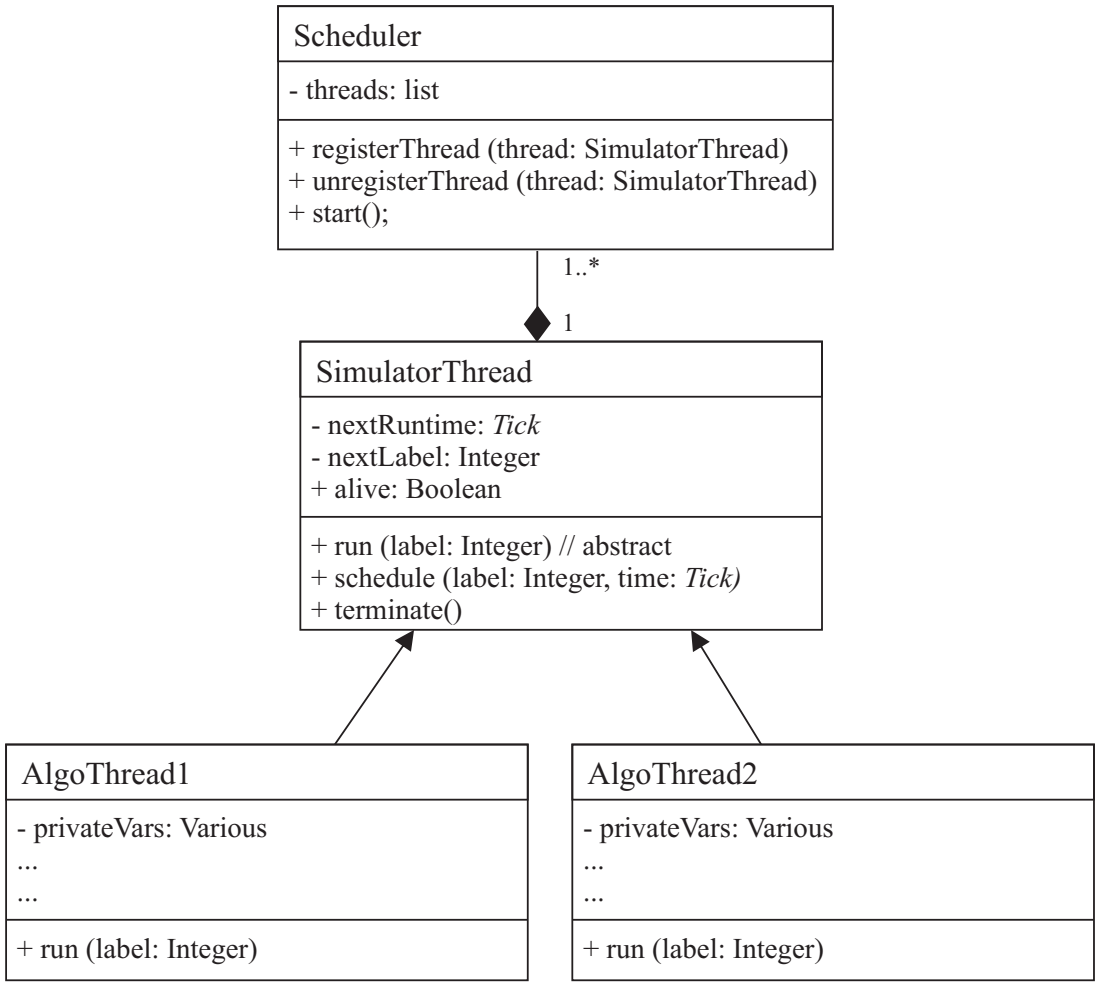


Figure 7.2: The class hierarchy of the simulator implementation that allows any code to be executed in the simulator.

ever, every algorithm that we proposed in this research re-uses threads. Therefore, for each different algorithm, we extended 'SimulatorThread' and created a thread which contains the implementation of the algorithm. The flexibility in the design to support custom code allows fast and easy modifications and improvements to the simulated program.

Locks and Waiting

'Read-write' locks are necessary to prevent more than one thread entering into a critical section of a program. In our proposed algorithm, the decision making process is executed in a 'read-write' lock. However, in our simulations we assume that the decision making process has a negligible runtime compared to the other tasks. It is a valid assumption since our experiments show that the decision making process is extremely fast and spends only a negligible fraction of the time compared the other long running tasks. Therefore, in our algorithm simulations we do not use any 'read-write' locks, although this type of locks can be still implemented in a similar manner to the barrier type locks described next.

For our algorithms, we use a conditional wait barrier (i.e. pthread_cond_wait in pthread library). This kind of barrier is used to make threads wait until an specific event occurs. All threads which wait on a barrier can be notified by another thread, which wakes up all the waiting threads. This kind of waiting is achieved in the simulator by scheduling a code block to a smaller *tick* than the current *tick*. An instance of a class named 'Lock' stores all waiting threads on a barrier. When it needs to start waiting on a barrier, a thread should register on the corresponding 'Lock' instance for the barrier. Once the 'Lock' is notified, 'run()' methods of all threads in the waiting list are called in the current *tick*.

7.2 The Parallelization Framework

Developing our final proposed algorithm is a result of number of improvements and changes over time. To facilitate these changes without much overhead of parallel programming, we developed a framework. The framework manages most of the complexities introduced by thread management and thread-safety. It also clearly separates

scheduling logic from thread management, ensuring a better decision making process with easier improvements and amendments.

The framework is developed as a distributed scheduling algorithm. Similar to the proposed algorithms in Section 5.2, each thread runs a decision making procedure to decide its behaviour. Each thread decides its next actions, when the thread becomes idle or waken up by another thread.

The decisions to be taken are:

- the next task to choose from the loading, comparing, waiting or terminating; and
- if it has decided to load or compare, the best GS to *load* or the best comparison to start.

Our aim is to separate the process of making these decisions from thread management and other concerns of the application such as race conditions. Therefore, we introduce a concept called *Rules*. The *Rules* are the output of the scheduling algorithm which is followed by each thread. Each thread consults the *Rules* to make a decision on their behaviour. The *Rules* act as a black-box which is aware of the current system status and makes decisions combining the current system status with its underlying instructions.

Figure 7.3 shows an illustration of the *Rules* as a black-box for decision making. The *Rules* constantly receives updates on the actions performed by the threads. It also has access to the system and dataset properties such as available memory capacity and sizes of the data items in memory. The *Rules* outputs three decisions each of which can be followed by a thread. Based on the action output by the *Rules*, the thread will ask for the properties related to the action. For instance, a thread instructed to start a *load* will consult the *Rules* to find out which data item it should *load*.

Figure 7.4 shows how a thread behaves under the guidance of *Rules*. Each thread in the system follows the same process simultaneously, as we discussed in the algorithm de-

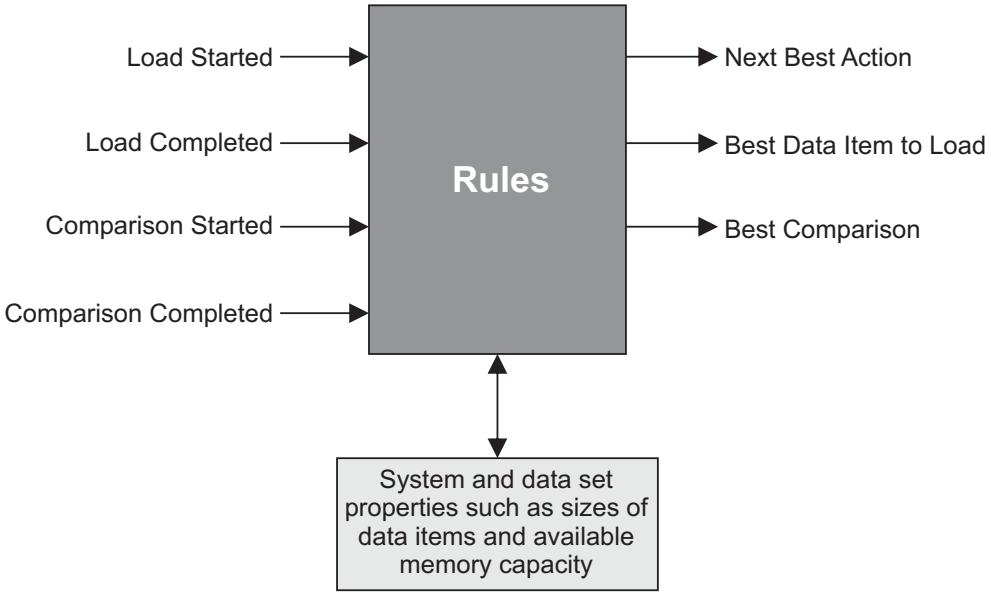


Figure 7.3: The *Rules* acts as a black box which takes current system status with changes to the status as inputs, and outputs decisions for the threads.

velopment. The regions drawn with continuous lines operates within a *read write lock*. Only one thread can enter to these regions to avoid race conditions during decision making. It is important to note that every call to *Rules* is made within the *read write lock*. Thus, the implementations within *Rules* do not have to deal with any thread-safety concerns. In addition, conditional waiting of threads is also handled external to the *Rules*. Therefore, *Rules* clearly separates decision making process from system's other concerns. As a result, modifications and improvements to the algorithms can be completed faster and easily without effecting the rest of the system.

In the implementations, the *Rules* is developed as an interface. Therefore, different implementations of the *Rules* can be dynamically changed according to the situation. For instance, a system with a sufficient memory to hold all data items, can utilize a different *Rules* implementation rather than using the same *Rules* designed for limited memory.

A remaining issue is how the unloads are handled in the framework, and we address it here. The unloading process is moved into *Rules*. Since decision making is done in a thread-safe manner and an unload has a negligible computational cost, unloading process is triggered within the *Rules*. It makes decision making process faster by allowing required unloads to perform quickly while making decisions. For instance, assume that there is a data item which does not relate to any comparison in memory. When *Rules* take the next decision, it can unload the data item immediately and ask the thread to start the next *load* rather than asking it to unload the item and then again asking it to *load* the next item.

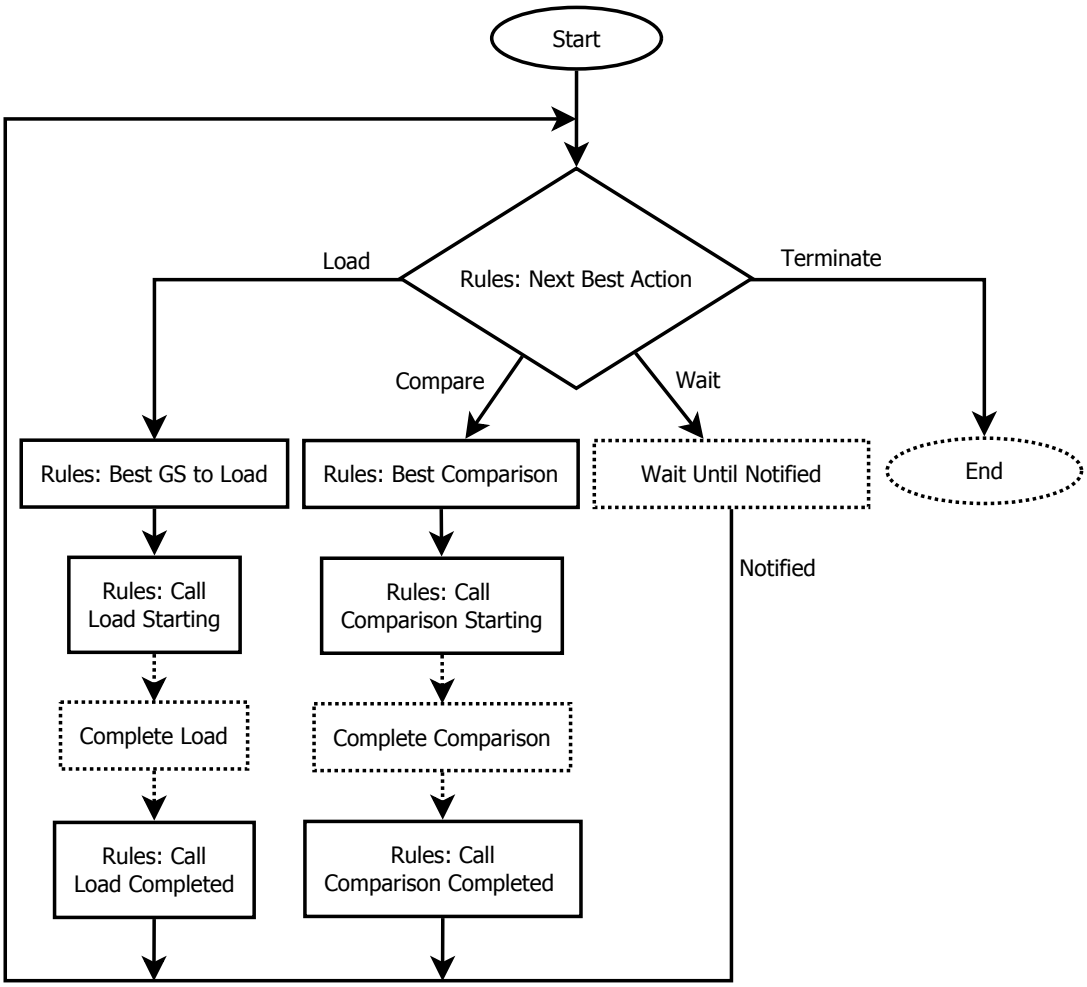


Figure 7.4: The flowchart of the framework which uses *Rules* as the decision maker. Every thread follows this procedure.

7.3 Visualizers

To analyse the behaviour of an algorithm or to troubleshoot, visualizing the behaviour is immensely helpful. We developed two unique visualizations for the DIAC problem. Those are the 'matrix' and 'thread' diagrams. We described these two types of diagrams in detail, in Section 5.2.1. While explaining our algorithms, we constantly used these two diagrams throughout the thesis. A screen-shot of the visualizer tool which was used to create these diagrams is shown in Figure 7.5.

The visualizer tool is based on our own log file format. The log file records: when a load, unload and comparison started and finished; details of each recorded task; and which thread completed which task. It also contains meta-data related to recorded instance, such as dataset name, computer name and memory limit used (these information is valuable when running the program in a computer). Once a log file is loaded into the visualizer tool, the information inside is used to create the diagrams. The waiting time of the threads in the thread diagram is derived by using the start and finish time of the tasks. In the tool, the diagrams can be further investigated by zooming and scrolling them. The labels on the task show the properties of the task for fast referral when there is sufficient room to display within the task. If the room is insufficient, it is hidden and can be viewed by hovering the mouse on the task. In the thread diagram, time between two events can be measured by using the time-stamp displayed while hovering the mouse over the diagram.

Another important feature in this tool is the ability to export these diagrams in postscript (ps) format. The exported diagram can be included in PDFs (Portable Document Format) as vector graphics. All thread and matrix diagrams in this thesis were generated by using this tool.

The visualizer not only helpful to analyse simulations but also helpful to analyse run-

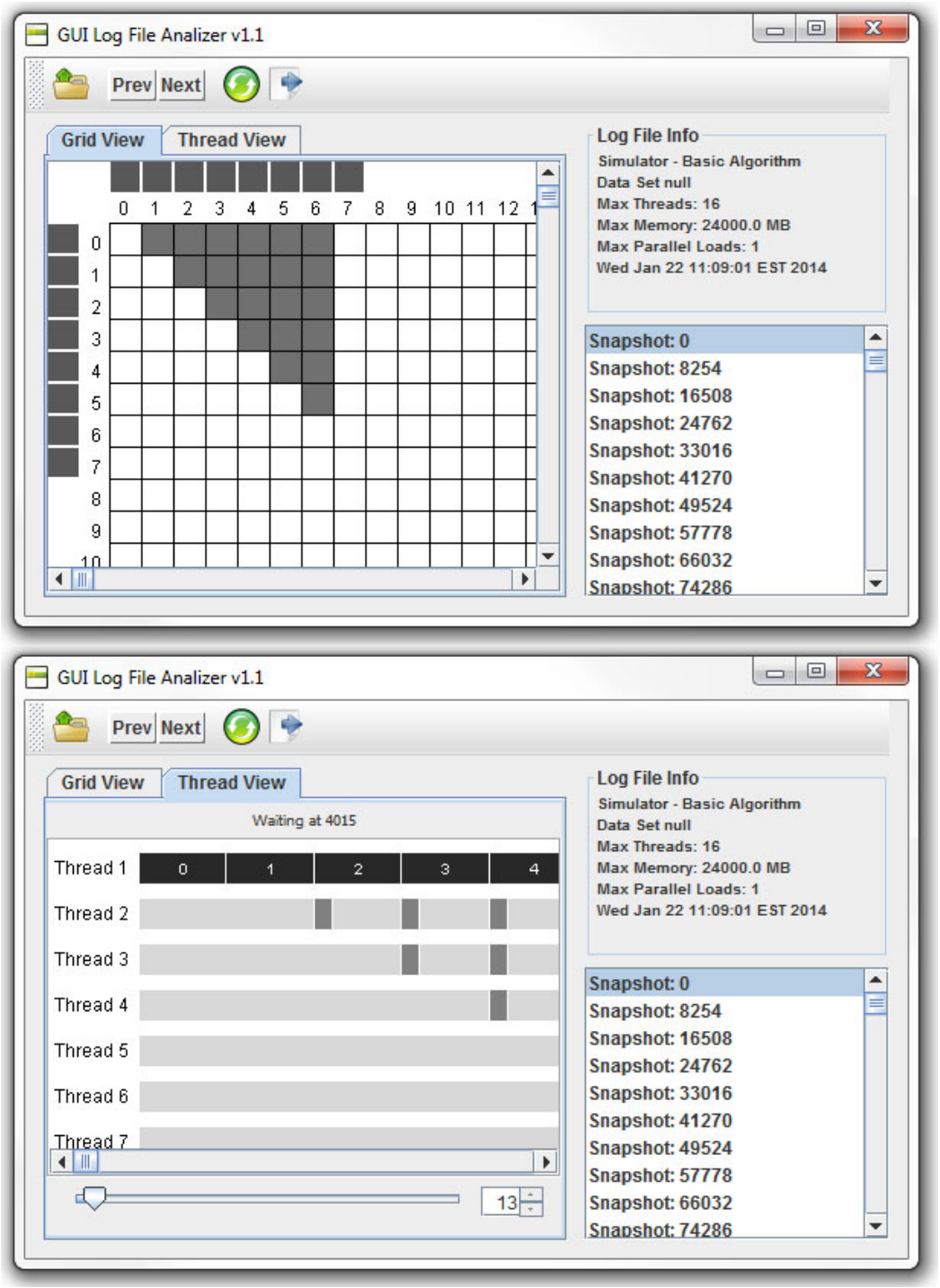


Figure 7.5: The visualizer tool that is used to support algorithm analytical and experimental phases in the research.

time implementations of our algorithms in real computer systems. The real implementations can also optionally generate log files in the runtime. These log files are extremely helpful to analyse the behaviour of the algorithm in various systems. The visualizer is designed to handle partial log files, so that it can assist in the detection of crashes, diagnosing the problems and identifying problems in algorithms and implementations.

7.4 Summary

In this chapter, we presented a novel, fast and lightweight simulator tool which exploits the advantages of the specific properties of the algorithms developed in this research. The simulator tool is flexible and can be used by other research with requirements similar to this research.

We also presented a parallel framework designed to separate decision making logic of scheduling in DIAC, from the thread-management and other system concerns. The framework facilitates the rapid implementation of the new proposed algorithms, and fast and convenient amendments to existing implementations. The major advantage of using the framework is that it reduces the overhead of implementation concerns such as race conditions and thread-safety, while developing the scheduler's logic. In addition, it also permits dynamic change of the decision making instructions (scheduling algorithm) in the runtime, based on the system and dataset properties.

Finally, we presented a unique visualization tool which is specifically designed to analyse the runtime behaviour of a DIAC based application. The simulator, the framework and the visualizer tools were extensively used in our research in the analytical and experimental phases.

Chapter 8

Conclusion

Data Intensive All-to-all Comparison (DIAC) problem is a frequent problem for many scientific and generic applications. We discussed many of these problems found in various fields such as bioinformatics, image processing and data mining in Chapter 2.

The increasing popularity of multi-core systems has emerged the need for solutions to parallelization of DIAC in shared memory multi-core computers. The nodes in many High Performance Computing clusters such as HPC in QUT [QUT, 2014], are shared-memory multi-processors systems. Even personal computers at the present time have multiple cores with a shared-memory. Therefore, solutions for parallelization of DIAC in shared-memory systems are needed to efficiently run a wide range of DIAC problems in these systems.

The ability of a program to operate within a certain memory limit is important in many scenarios. Mainly, typical shared-memory systems have a limited memory capacity for data. From the examples used for the experiments in Chapter 3 and 6, it is evident that some DIAC programs face this limited memory capacity problem with large data sets. In addition, many High Performance Computing facilities such as HPC in

QUT [QUT, 2014], requires the users to specify the maximum memory requirement of their programs. When a program exceeds the specified limit it may be killed by the batch processing software. Highly data intensive applications like DIAC poses a challenge to the users of these systems to exploit reasonable parallelism in their programs while staying within the specified memory limit.

Slow speed of the virtual memory for computationally intensive applications is another reason for staying within the system's physical memory. As we experimentally analysed in Section 3.5.4, the performance of an application can be dramatically hindered with the involvement of virtual memory. Although, virtual memory is an excellent option for handling excessive memory usage over the physical memory capacity, this negative effect can painfully hinder performance of computationally intensive applications. Therefore, limiting the memory usage of an application is a better option than letting the program slip into the virtual memory.

The parallelization of DIAC is a difficult problem to solve in limited memory. Even in its simplest form (i.e. with uniform sizes and runtime) there are huge number of combinatorial solutions (over the life span of the application) in limited memory conditions. As shown in Section 4.1.2, there are large number of different paths to choose at every system state throughout the life span of the execution. The main reason for these huge number of combinations in solutions is the data item sharing pattern in DIAC and the limited memory.

As we discussed in Chapter 2 there are some attempts to parallelize DIAC. Only few among these attempts target parallelization of DIAC in shared-memory systems. None of them have attempted parallelization of memory-constrained DIAC. There exist only one research report targeting to solve the memory handling problem in DIAC in limited memory conditions. However, they only provide theoretical complexity analyses of solving the problem and propose solutions for unlimited memory scenario.

We also investigated the effectiveness of existing parallelization techniques to solve the memory-constrained DIAC problem in Chapter 4 by extending these existing techniques to solve the problem of parallelization of DIAC in limited memory. However, due to the complexity introduced by memory limitations, these existing techniques rendered ineffective.

To deal with the complicated parallelization problem, we divided and conquered it in two manageable phases. First, we focused only on memory management problem in Chapter 3, leaving the parallelization aside. Chapter 3 successfully developed a memory management algorithm which significantly increased the speed of our benchmark applications compared to the existing algorithms. Then, the second phase that is for parallelization of DIAC was successfully completed in Chapter 5 and 6 based on the memory management strategies in the first phase. In this phase a novel parallelization technique was developed. Based on this parallelization technique a new algorithm to parallelize DIAC in limited memory was developed which also efficiently manages memory. We designed, implemented and evaluated the proposed algorithm in a simulation environment and in real computing platforms. This algorithm dramatically improved the speed of our benchmark applications in comparison to their sequential counterparts.

A simulation environment, parallel framework and visualization tool were specifically designed to be used in our research in Chapter 7.

The summary of the research contributions is as follows. The major contributions are listed in detail in Section 8.1 and the minor and incremental contributions related to the major contribution are listed in Section 8.2.

8.1 Major Contributions

The following are the three major contributions of this thesis.

- In the first phase of the research (in Chapter 3), we proposed a novel efficient memory management (paging) algorithm to increase performance by minimizing the *loads* in a DIAC, for uniformly sized data items. We demonstrated this algorithm as one of the optimum algorithms up to $N = 9$ for memory-constrained DIAC for all possible M values, by using a brute-force method that tries all possible combinations. Then we extended this algorithm to a scalable memory management algorithm which can handle non-uniformly sized data items and the variations of free memory due to other background processes. This scalable algorithm was experimentally proven to be more efficient and accurate than existing algorithms. The accuracy refers to not over using memory than a specified limitation. The experiments showed that our algorithm improved the speed of our benchmark applications from 7% to 31.9 times compared with existing methods to manage memory.
- In the second phase of the research, a novel parallelization technique was developed to exploit the advantage of prior knowledge of the targeted problem. This technique was developed in the process of developing a parallelization algorithm to solve our DIAC problem. This parallelization technique combines dynamic scheduling with static scheduling and utilizes a pattern developed based on the known properties of the targeted problem abstraction. Due to the combination of dynamic scheduling guided by a pattern based static scheduling strategy, the computational complexity and the time spent on scheduling is kept at a minimum level while still keeping the ability to adjust dynamically to variations in the application runtime. In comparison to the existing parallelization techniques (evaluated in Chapter 4), our parallelization technique is capable of minimizing the runtime overhead of scheduling algorithms to solve complex parallelization

problems such as DIAC, where significant prior knowledge of the problem abstraction is available (e.g. knowledge on data dependencies and tasks). Based on our parallelization technique, a new algorithm (Algorithm 7) was developed for parallel execution of memory-constrained DIAC in shared-memory multi-core computers. The algorithm scales well in various multi-core and multi-processor shared memory platforms as shown by the experiments. The algorithm dynamically adjusts well, even in the absence of accurately estimated comparison and *load* times, as shown by the experimental results. The experiments also show that the algorithm achieves near optimum processor utilization for completing comparisons when sufficient memory is present. The algorithm dramatically increases the speed of existing bioinformatics computing applications in comparison with their sequential counterparts while efficiently managing the memory. For instance, in an octa-core system, a test of our algorithm for a benchmark example shows a runtime of two weeks reduced to less than two days.

- To assist evaluating the performance of the proposed algorithms, a simulator and a parallel framework was developed. The simulation environment was developed with flexibility in a such a way that it can be adapted to simulate parallelization algorithm with similar properties to our algorithms. The simulator exploits the advantage of assumptions in our algorithms which are continuous processor task affiliation and uninterrupted task execution in a processor. Therefore, it is light-weight in runtime and extremely fast in simulating long running applications. The simulator is also capable of emulating theoretical behaviour of the algorithms such as zero computational cost for scheduling. This feature is useful for theoretical validations. The framework was designed and implemented for rapid implementation of different parallelization strategies for DIAC in shared-memory systems. The framework clearly separates implementations concerns such as thread-safety from the decision making process in scheduling. This framework also allows to change the parallelization algorithm (i.e. scheduling strategies) dynamically in the runtime based on the system and data

set properties.

8.2 Minor and Incremental Contributions

The following is a list of minor and incremental contributions related to the major contributions in the same order as they appear in the thesis.

- A close-fitting theoretical lower bound was derived for the minimum number of *loads* required to complete a DIAC in limited memory for uni-processor systems.
- For the algorithm developed for memory management in DIAC for uni-processor systems, optimum parameters for its best performance were theoretically derived. Five theorems were developed to mathematically derive the optimum parameters for the algorithm.
- The problem of parallelizing memory-constrained DIAC is modelled as a resource constrained task scheduling problem. The reason for modelling the problem as a resource constrained scheduling problem is that the memory, processors and disk constraints as resources limit the execution of comparisons. The optimization problem to be solved in the model is scheduling all tasks in a DIAC within a minimum time-span while meeting all resource constraints and task dependencies. After modelling, the scheduling problem was identified as deviating from traditional scheduling problems. The deviation occurs since the *load* operation holds memory resources even after completion, and a different operation (i.e. an unload) is required to release the held memory resources.
- We examined a number of generic and specific existing parallelization techniques in the literature. Out of them, four techniques were selected, namely DAG based, self adjusting dynamic scheduling (SADS), heuristic functions based scheduling and local search. The resource constrained scheduling based model was extended to develop four new models to be compatible with each of the four

selected techniques. None of the methods are sufficiently effective in solving our DIAC problem due to one or more of the following three reasons; 1) their excessive computational cost, 2) infeasibility to present memory constraints and 3) producing incomplete solutions. However, the heuristic functions based approach was heavily utilized as a guidance while developing our algorithm. For the local search technique we developed a new model by combining existing generic memory management techniques and Simulated Annealing (SA). The aim was to build a reference schedule by running a search algorithm for a long time, to be utilized in the algorithm development process. The approach was successful in building a reference schedule.

- A theoretical upper bound for the maximum gain by using parallel execution was derived for the memory-constrained DIAC. The derivation of the upper-bound shows that the limitation of the memory always limits the maximum speed gain that can be achieved in a DIAC, regardless of the available processors. The upper bound also explains the limitations of parallel gain in the proposed parallel algorithms.
- A fast mathematical approach was developed for estimating the parameters for the optimum result of our parallel algorithms. The approach addressed most frequent scenarios found in general computing platforms. For other scenarios, simulations are used to estimate the optimum parameters.
- Two visualization diagrams called ‘matrix’ and ‘thread’ diagrams were developed to analyse the behaviour of the algorithm in the simulator and real computing platforms. These diagrams together with the visualization tool was immensely helpful for identifying the problems, finding possible improvements and validating the intended behaviour of the algorithms. The visualization tool generates ‘matrix’ and ‘thread’ diagrams based on a specific log file format generated by a DIAC application and enables the users to examine the two diagrams easily.

8.3 Limitation and Future Work

This research only focuses on shared-memory systems. Shared-memory systems have the advantage of data sharing among the cores and faster communication between cores (processors). We successfully demonstrated that shared-memory computers with sufficient memory and large number of cores can significantly increase the speed of *Data Intensive All-to-all Comparison (DIAC)* problem using our algorithms. However, our solution is limited to the number of cores which reside in a single node of a large computing facility. In some scenarios with huge data sets, it could be beneficial to distribute computation to multiple nodes despite the communication overhead of distributing data and repeated preprocessing of data items.

Moreover, there can be even larger data sets that does not fit into a single node's disk storage. On the other hand, the speed gain from the cores in a single shared memory computer may not be sufficient. Therefore, as the next leap of this research, shared-distributed memory computers must be supported. As described in Section 2.6, shared-distributed memory systems are a network of shared-memory computers connected through an external network. Most high performance computing facilities at the present time have this architecture. So, our parallel algorithm is still useful in these kinds of systems to handle memory management and parallel execution within the nodes. In these situations, frameworks like Hadoop [Kurazumi et al., 2012] can be incorporated with our memory management solution to take the full advantage of shared-distributed memory systems.

Appendix A

Estimating Load Times and Composition Vector's Size

This appendix presents some of the bioinformatics composition vector specific analyses on predicting the *composition vector* size and *load* time from the disk. This appendix is based on the composition vector method proposed by Yu et al. [2010a].

The *load* time estimations of a *composition vector* is two fold as calculating the composition vector from the scratch and reading a pre-calculated *composition vector* from the disk. First we look into the estimation of calculating a *composition vector*. The preprocessing algorithm consists of five considerably time-consuming loops. Table A.2 describes each step and its time consumption estimations. The total time is the sum of the time spent in all of these major steps. All the notations used in the Table A.2 are described in Table A.1.

As seen in the Table A.2, execution times for three steps of the preprocessing are predictable based on the length of a gene sequence. Tthe length of the genomic sequence is the only parameter which can be retrieved without spending significant amount of

Table A.1: Notations used in Table A.2.

K:	Order of comparison
A:	Alphabet size (4 or 20)
S:	Size of the genome file
C_i:	Arbitrary constants
V:	Number of non-zero values in the T vector

time. In the runtime for a large dataset, by only analysing few *composition vector* calculations (profiling), C_0, C_1 and C_2 can be easily determined as they are constants for a system.

However, the last two steps mainly depends on the context of the gene sequence. Figure A.1 shows the variation of time spent on generating T Vector versus the file size of the sequences. The two circled sequences are nearly similar in file size, being 98 kb and 100 kb in size. However, the time spent in T Vector generation is largely different even if the loop runs A^K times every time. Since the loop in the T Vector generation step has the same number of iterations irrespective of the file size, the loop was profiled for the two particular gene sequences.

Figure A.2 shows the comparison of profiling of T-Vector generation of above two gene sequences. It shows the Line 188 has a major difference in the execution time. Since, the number of times the line 188 is executed depends on the content of the sequence, it has made a significant difference of the time consumed for this step. The final loop iterates number of times equals to the non-zero values in the T Vector. Therefore the time spent on this step depends on the content of the genomic sequence. Due to the incapability of predicting the time consumption of the last two steps, the time spent to calculate a composition vector cannot be quickly estimated accurately.

Table A.2: Steps in calculating a *composition vector* and complexity analyses of each step. Please see Table A.1 for the notations.

Step	**Time Estimation**	**Complexity** with S	**Complexity** with K	**Description**
Initializing Vectors	$C_0(A^K + A^{K-1})$	$O(1)$	$O(k^n)$	Creating and initializing long vectors with a length. Constant for calculations with a same order of calculation.
Reading File	$C_1 \times S$	$O(n)$	$O(n)$	Reading file is linear to the file size and experimentally verified.
Second Vector Totals	$C_2 \times A^{K-1}$	$O(1)$	$O(k^{(n-1)})$	Constant for a same order of calculation.
T Vector Generation	$C_3 \times A^K$	$O(1)$	$O(k^n)$	Depends on the content of the gene sequence. Highly unpredictable. Described in more details later.
Final Vector Generation	$C_4 \times V$	$O(?)$	$O(?)$	Linear to the non-zero values produced in the T Vector and experimentally verified. Depends on the composition of the gene sequence.

The accurate sizes of the *composition vectors* in memory is essential for accurate memory management. If the size of the *composition vector* can be predicted by using the genomic sequence significantly faster than calculating the *composition vector*, there is no requirement to pre-calculate all *composition vectors* to determine their size before starting the comparison. So, we look into the composition vector method to find out whether any quick and accurate estimation is possible or not. Table A.2 lists the steps of *composition vector* calculation. The step 'T Vector Generation' depends on the content of the genomic sequence. Therefore, the number of non-zero values depends on the content of the genomic sequence which cannot be estimated accurately. Therefore, the length of the final *composition vector* depends on the content of the genomic sequence and highly unpredictable.

Figure A.3 shows how the *load* times varies with the size of the *composition vector* when loaded from the disk. We tested two computer setups with Data Set 2 listed in Section 6.4.2. It can be seen from the graph that, the loading times are mostly unpredictable and not distributed linearly. However, when size increases the *load* time generally increases.

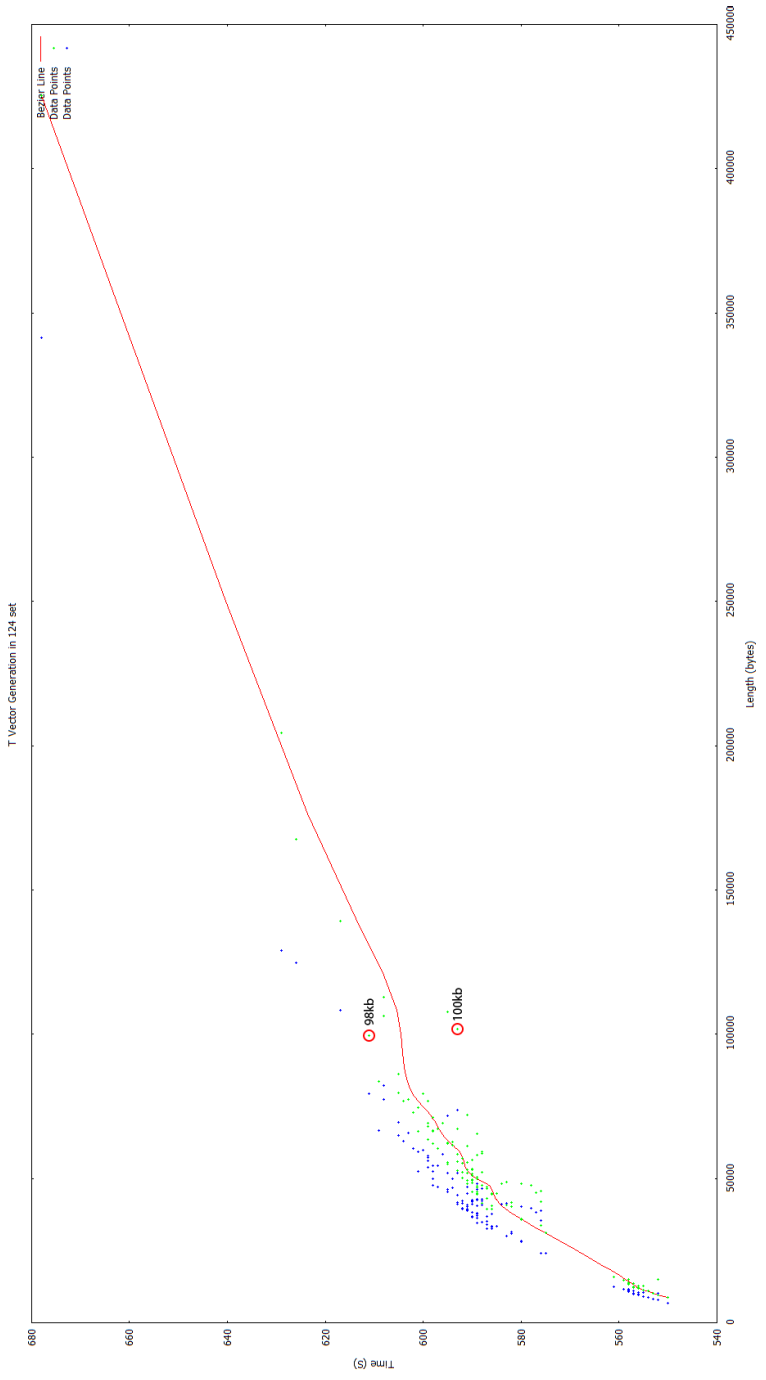


Figure A.1: Significant difference in time of nearly similar sized genomes in T Vector calculation

Source | Assembly

Line	Source	CPU Time:Difference 100kb	CPU Time:Result 1 100kb	CPU Time:Result 2 98kb	Instructions Retired:Difference	Instructions ...	Instructions ...
163							
164							
165	start_timing(&(log_time_array[bacteria_number]->t_vector));						
166	count = 0;						
167	double* t = new double[M];						
168	for (long i = 0; i < M; i++) {	-25.401ms	0.226s	0.251s	-122,000,000	496,000,000	618,000,000
169	double p1 = second_div_total[i_div_aa_number];	7.353ms	0.417s	0.410s	138,000,000	2,964,000,000	2,826,000,000
170	double p2 = one_l_div_total[i_mod_aa_number];	-11.364ms	0.044s	0.055s	-62,000,000	178,000,000	240,000,000
171	double p3 = second_div_total[i_mod_M1];						
172	double p4 = one_l_div_total[i_div_M1];						
173	double stochastic = (p1 * p2 + p3 * p4) * total_div_2;		1.322s	1.322s	70,000,000	6,726,000,000	6,656,000,000
174							
175	if (i_mod_aa_number == AA_NUMBER - 1) {	-101.604ms	0.643s	0.745s	-260,000,000	4,360,000,000	4,620,000,000
176	i_mod_aa_number = 0;	-30.080ms	0.077s	0.107s	-120,000,000	440,000,000	560,000,000
177	i_div_aa_number++;		0.027s	0.027s	36,000,000	148,000,000	112,000,000
178	} else						
179	i_mod_aa_number++;	-1.337ms		0.001s			
180							
181	if (i_mod_M1 == M1 - 1) {	-13.369ms	0.291s	0.305s	162,000,000	1,844,000,000	1,682,000,000
182	i_mod_M1 = 0;						
183	i_div_M1++;						
184	} else						
185	i_mod_M1++;	-30.749ms	0.181s	0.212s	78,000,000	1,236,000,000	1,158,000,000
186							
187	if (stochastic > EPSILON) {	26.738ms	0.120s	0.094s	-52,000,000	848,000,000	900,000,000
188	t[i] = (vector[i] - stochastic) / stochastic;	-638.369ms	1.447s	2.086s	-596,000,000	1,840,000,000	2,436,000,000
189	count++;						
190	} else						
191	t[i] = 0;	10.695ms	0.884s	0.874s	526,000,000	4,672,000,000	4,146,000,000
192	}						
193							
194	delete second_div_total;						
195	delete vector;						
196	delete second;						

Figure A.2: Profiling of T-Vector generation step of two nearly similar sized gene sequences

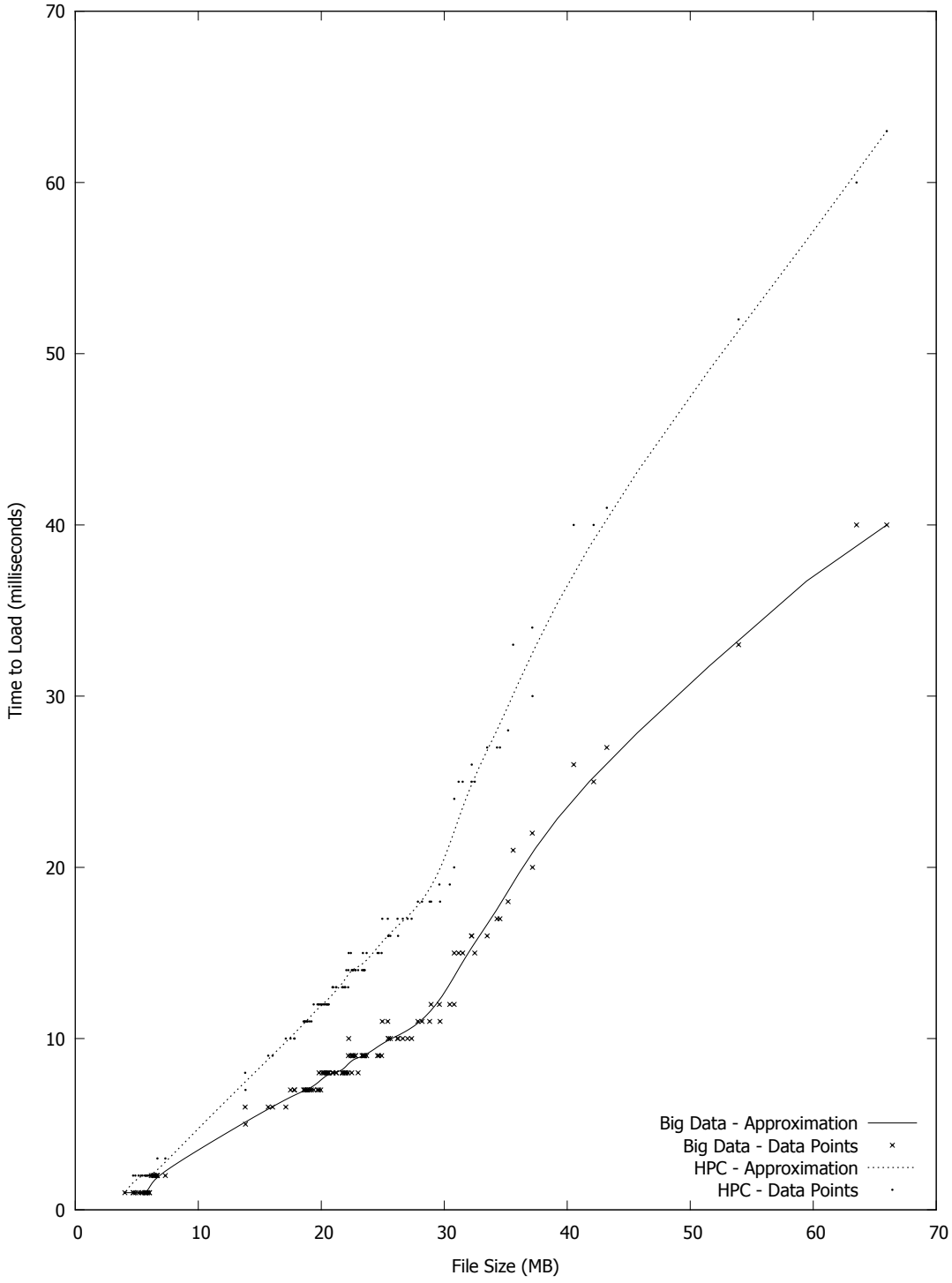


Figure A.3: Variation of *load* times when *composition vectors* are loaded from the disk in Data Set 2

Bibliography

Abdelkader, D. M. and Omara, F. (2012). Dynamic task scheduling algorithm with load balancing for heterogeneous computing system. *Egyptian Informatics Journal*, 13(2):135 – 145.

Agrawal, R., Faloutsos, C., and Swami, A. (1993). Efficient similarity search in sequence databases. pages 69–84. Springer Verlag.

Aho, A. V., Denning, P. J., and Ullman, J. D. (1971). Principles of optimal page replacement. *Journal of ACM*, 18(1):80–93.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403 – 410.

Amalarethinam, D. G. and Mary, G. J. (2011). A new dag based dynamic task scheduling algorithm (DYTAS) for multiprocessor systems. *International Journal of Computer Applications*, 19(8):24–28.

Amalarethinam, D. I. G. and Mary, G. J. J. (2010). A survey : Dynamic task scheduling in multiprocessor and the swift embryonic world of parallel computing. *International Journal of Algorithms, Computing and Mathematics*, 3:53–60.

Amdahl, G. M. (2013). Computer architecture and Amdahl's law. *Computer*, 46(12):38–46.

Austin, T., Larson, E., and Ernst, D. (2002). SimpleScalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67.

Aven, O., Boguslavsky, L., and Kogan, Y. (1976). Some results on distribution-free analysis of paging algorithms. *IEEE Transactions on Computers*, C-25(7):737–745.

Barton, G. J. (1990). Protein multiple sequence alignment and flexible pattern matching. *Methods in enzymology*, 183:403–428.

Berman, F., Casanova, H., Legrand, A., and Zagorodnov, D. (1999). Using simulation to evaluate scheduling heuristics for a class of applications in grid environments. Technical report.

Braun, T. D., Siegel, H. J., Beck, N., Blni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810 – 837.

Bridges, M. J. (2008). *The VELOCITY compiler: extracting efficient multicore execution from legacy sequential codes*. PhD thesis, Princeton University, Princeton, NJ, USA.

Casanova, H., Legrand, A., Zagorodnov, D., and Berman, F. (2000). Heuristics for scheduling parameter sweep applications in grid environments. In *proceedings of 9th Heterogeneous Computing Workshop (HCW 2000), 2000.*, pages 349–363.

Ceze, L., Strauss, K., Almasi, G., Bohrer, P. J., Brunheroto, J. R., Cascaval, C., Castanos, J. G., Lieber, D., Xavier, Martorell, X., Moreira, J. E., Sanomiya, A., and Schenfeld, E. (2003). Full Circle: simulating linux clusters on linux clusters. In *proceedings of the Fourth LCI International Conference on Linux Clusters: The HPC Revolution 2003*, Las Vegas.

Chan, R. H., Chan, R. H., and Chan, R. H. (2010). Composition vector method for phylogenetics - a review. In *the Ninth International Symposium on Operations Research and Its Applications (ISORA10)*, pages 14–20, Chengdu-Jiuzhaigou, China.

Corpet, F. (1988). Multiple sequence alignment with hierarchical clustering. *Nucleic Acids Research*, 16(22):10881–10890.

CVTree (2011). Composition vector tree version 2. http://tlife.fudan.edu.cn/cvtree/.

Date, S., Kulkarni, R., Kulkarni, B., Kulkarni-Kale, U., and Kolaskar, A. (1993). Multiple alignment of sequences on parallel computers. *Computer applications in the biosciences (CABIOS)*, 9(4):397–402.

Edgar, R. C. and Batzoglou, S. (2006). Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368 – 373.

Feng, D.-F. and Doolittle, R. (1987). Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360.

Feng, D. F. and Doolittle, R. F. (1990). Progressive alignment and phylogenetic tree construction of protein sequences. *Methods Enzymol*, 183:375–387.

Feng, J., Chen, Y., and Summerville, D. (2010). Eeo: an efficient mds-like raid-6 code for parallel implementation. In *Sarnoff Symposium, 2010 IEEE*, pages 1–5.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948 –960.

Geng, X., Xu, G., and Zhang, Y. (2010). Dynamic load balancing scheduling model based on multi-core processor. In *2010 Fifth International Conference on Frontier of Computer Science and Technology (FCST)*, pages 398–403, Changchun, China.

Giersch, A., Robert, Y., and Vivien, F. (2003). Scheduling tasks sharing files on heterogeneous clusters. In Dongarra, J., Laforenza, D., and Orlando, S., editors, *Recent*

Advances in Parallel Virtual Machine and Message Passing Interface, volume 2840 of *Lecture Notes in Computer Science*, pages 657–660. Springer Berlin Heidelberg.

Giersch, A., Robert, Y., and Vivien, F. (2004). Scheduling tasks sharing files on heterogeneous master-slave platforms. In *proceedings of 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004.*, pages 364–371, A Coruna, Spain.

Gotoh, O. (1993). Optimal alignment between groups of sequences and its application to multiple sequence alignment. *Computer applications in the biosciences (CABIOS)*, 9(3):361–370.

Hamidzadeh, B., Kit, L. Y., and Lilja, D. (2000). Dynamic task scheduling using online optimization. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1151–1163.

Hamidzadeh, B. and Lilja, D. (1996). Dynamic scheduling strategies for shared-memory multiprocessors. In *proceedings of the 16th International Conference on Distributed Computing Systems*, pages 208–215, Hong Kong.

Han, G. S., Yu, Z. G., Anh, V., Krishnajith, A. P. D., and Tian, Y.-C. (2013). An ensemble method for predicting subnuclear localizations from primary protein structures. *PLoS ONE*, 8(2):e57225.

Hao, B., Qi, J., and Wang, B. (2003). Prokaryotic phylogeny based on complete genomes without sequence alignment. *Modern Physics Letters*, 2:14.

Havlak, P., Chen, R., Durbin, K. J., Egan, A., Ren, Y., Song, X.-Z., Weinstock, G. M., and Gibbs, R. A. (2004). The atlas genome assembly system. *Genome Research*, 14:721–732.

Higgins, D. G. and Sharp, P. M. (1988). CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237 – 244.

Hill, J., Hambley, M., Forster, T., Mewissen, M., Sloan, T. M., Scharinger, F., Trew, A., and Ghazal, P. (2008). SPRINT: A new parallel framework for R. *BMC Bioinformatics*, 9:558.

Huang, X., Wang, J., ALuru, S., Yang, S.-P., and Hillier, L. (2003). PCAP: a whole-genome assembly program. *Genome Research*, 13:2164–2170.

Jaap and Heringa (1999). Two strategies for sequence comparison: profile-preprocessed and secondary structure-induced multiple alignment. *Computers & Chemistry*, 23(34):341 – 364.

Jaap and Heringa (2002). Local weighting schemes for protein multiple sequence alignment. *Computers & Chemistry*, 26(5):459 – 477.

Katoh, K., Misawa, K., Kuma, K.-i., and Miyata, T. (2002). MAFFT: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic Acids Research*, 30(14):3059–3066.

Katoh, K. and Toh, H. (2010). Parallelization of the MAFFT multiple sequence alignment program. *Oxford Journals: Bioinformatics*, 26:1899–1900.

Kleinjung, J., Douglas, N., and Heringa, J. (2002). Parallelized multiple alignment. *Bioinformatics*, 18(9):1270–1271.

Krishnajith, A., Kelly, W., Hayward, R., and Tian, Y.-C. (2013). Managing memory and reducing I/O cost for correlation matrix calculation in bioinformatics. In *proceedings of the 2013 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB),*, pages 36–43, Singapore.

Krishnajith, A. P. D., Kelly, W., and Tian, Y.-C. (2014). Optimizing I/O cost and managing memory for composition vector method based correlation matrix calculation in bioinformatics. *Current Bioinformatics*. In press.

Kurazumi, S., Tsumura, T., Saito, S., and Matsuo, H. (2012). Dynamic processing slots scheduling for i/o intensive jobs of hadoop mapreduce. In *2012 Third International Conference on Networking and Computing (ICNC)*, pages 288–292.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719.

Lee, Y. C. and Zomaya, A. (2006). Data sharing pattern aware scheduling on grids. In *ICPP 2006 International Conference on Parallel Processing*, pages 365–372, Columbus, Ohio, USA.

Li, Z.-S., Liu, D.-W., and Bi, H.-J. (2008). CRFP: a novel adaptive replacement policy combined the lru and lfu policies. In *IEEE 8th International Conference on Computer and Information Technology Workshops, 2008. CIT Workshops 2008.*, pages 72–79.

Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828.

Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., and Werner, B. (2002). Simics: a full system simulation platform. *Computer*, 35(2):50–58.

Manber, U. (1994). Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 2–2, Berkeley, CA, USA. USENIX Association.

Mckinley, P. K. and Trefftz, C. (1993). MultiSim: a simulation tool for the study of large-scale multiprocessors. In *proceedings of the 1993 International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Networks (MASCOTS, 1993)*, pages 57–62.

Meng, Z., Lin, Y., Kang, Y., and Yu, Q. (2013). A parallel programming pattern based on directed acyclic graph. *Applied Mechanics and Materials*, 303-306:2165–2169.

Miller, W. (1993). Building multiple alignments from pairwise alignments. *Computer applications in the biosciences : CABIOS*, 9(2):169–176.

Moretti, C., Bui, H., Hollingsworth, K., Rich, B., Flynn, P., and Thain, D. (2010). All-pairs: An abstraction for data-intensive computing on campus grids. *IEEE Transactions on Parallel and Distributed Systems*, 21(1):33–46.

Mueen, A., Nath, S., and Liu, J. (2010). Fast approximate correlation for massive time-series data. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 171–182, New York, NY, USA. ACM.

Notredame, C., Higgins, D. G., and Heringa, J. (2000). T-coffee: a novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205 – 217.

Omara, F. A. and Arafa, M. M. (2010). Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing*, 70(1):13 – 22.

Pacheco, P. (1997). *Parallel Programming With MPI*. Morgan Kaufmann Publishers.

Pekurovsky, D., Shindyalov, I. N., and Bourne, P. E. (2004). A case study of high-throughput biological data processing on parallel platforms. *Bioinformatics*, 20(12):1940–1947.

Pertel, M. J. (1992). A simple simulator tool for multicomputer networks. Caltech Computer science technical report.

Phillips, P., Flynn, P., Scruggs, T., Bowyer, K., Chang, J., Hoffman, K., Marques, J., Min, J., and Worek, W. (2005). Overview of the face recognition grand challenge. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 947–954 vol. 1.

Prakash, S. and Bagrodia, R. (1998). MPI-SIM: using parallel simulation to evaluate mpi programs. In *Simulation Conference Proceedings, 1998*, pages 467–474.

Qi, J., B., W., and Hao, B. (2004). Whole proteome prokaryote phylogeny without sequence alignment: A k-string composition approach. *Journal of Molecular Evolution*, 58(1):1–11.

QUT (2014). High performance computing & research support. http://www.itservices.qut.edu.au/researchteaching/hpc/.

Rajasekaran, S., Thapar, V., Dave, H., and Huang, C.-H. (2005). Randomized and parallel algorithms for distance matrix calculations in multiple sequence alignment. *Journal of Clinical Monitoring and Computing*, 19:351–359.

Ramamritham, K., Stankovic, J., and Shiah, P.-F. (1990). Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194.

Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A. (1993). The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. *SIGMETRICS Perform. Eval. Rev.*, 21(1):48–60.

Rizk, N. J. (2005). *Parallel and Evolutionary Approaches to Computational Biology*, pages 1–28. John Wiley & Sons, Inc.

Schuler, G. D., Altschul, S. F., and Lipman, D. J. (1991). A workbench for multiple alignment construction and analysis. *Proteins: Structure, Function, and Bioinformatics*, 9(3):180–190.

Shroff, P., Watson, D., Flann, N., and Freund, R. (1996). Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments. In *Heterogeneous Computing Workshop*.

Stein, C. M. (1981). Estimation of the mean of a multivariate normal distribution. *The Annals of Statistics*, 9(6):pp. 1135–1151.

Steinbiss, S. and Kurtz, S. (2012). A new efficient data structure for storage and retrieval of multiple biosequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(2):345 –357.

Sudhakar, C. (2006). Restructuring and extensible simulator for shared memory and message passing parallel programs. In *International Conference on Advanced Computing and Communications, 2006 (ADCOM, 2006)*, pages 2–7.

Tao, T. (2012). Single letter codes for nucleotides. Accessed on Mar 2012. http://www.ncbi.nlm.nih.gov/staff/tao/tools/tool_lettercode.html.

Thompson, J. D., Higgins, D. G., and Gibson, T. J. (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680.

Trelles, O. (2011). On the parallelization of bioinformatic applications. http://www.ac.uma.es/ ots/papers/survey.pdf.

Trelles, O., Andrade, M. A., Valencia, A., Zapata, E. L., and Carazo, J. M. (1998). Computational space reduction and parallelization of a new clustering approach for large groups of sequences. *Bioinformatics*, 14(5):439–451.

Ullman, J. D. (1975). NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393.

Wang, W. (2009). *Composition Vector Methods for Phylogeny*. PhD thesis, The Chinese University of Hong Kong.

Waterman, M. S. (1995). *Introduction to computational biology*. Chapman & Hall/CRC Press.

Wu, J.-J., Chou, E.-J., and Liu, P. (2009). Computation and communication schedule

optimization for data-sharing tasks on uniprocessor. *Journal of Systems Architecture*, 55(79):363 – 372.

Xiangbin, Z. and Shiliang, T. (2003). An improved dynamic scheduling algorithm for multiprocessor real-time systems. In *proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2003. (PDCAT'2003)*, pages 710–714, Chengdu, China.

Xu, Z. and Hao, B. (2009). CVTree update: a newly designed phylogenetic study platform using composition vectors and whole genomes. *Nucleic Acids Res.*, 37:174–178.

Yu, W., Gao, Q., and Panda, D. (2006). Adaptive connection management for scalable mpi over infiniband. In *20th International Parallel and Distributed Processing Symposium, 2006. IPDPS 2006.*, pages 25–29.

Yu, Z., Zhou, L., Anh, V., Chu, K., Long, S., and Deng, J. (2005). Phylogeny of prokaryotes and chloroplasts revealed by a simple composition approach on all protein sequences from complete genomes without sequence alignment. *Journal of Molecular Evolution*, 60:538–545.

Yu, Z.-G., Chu, K. H., Li, C. P., Anh, V., Zhou, L.-Q., and Wang, R. W. (2010a). Whole-proteome phylogeny of large dsDNA viruses and parvoviruses through a composition vector method related to dynamical language model. *BMC Evolutionary Biology*, 10.

Yu, Z.-G., Zhan, X.-W., Han, G.-S., Wang, R. W., Anh, V., and Chu, K. H. (2010b). Proper distance metrics for phylogenetic analysis using complete genomes without sequence alignment. *International Journal of Molecular Sciences*, 11:1141–1154.

Zhi, Y., Liu, Y., Jiao, L., and Zhang, P. (2010). A parallel simulator for large-scale parallel computers. In *9th International Conference on Grid and Cooperative Computing (GCC)*, pages 196–200, Nanjing, Jiangsu, China.

Zhu, Y., Shasha, D., and Shasha, Y. Z. D. (2002). StatStream: statistical monitoring of thousands of data streams in real time. In *proceedings of 28th International Conference on Very Large Data Bases*, pages 358–369, Hong Kong, China.