

https://dspace.org/

Massive parallelism for combinatorial problems by hardware acceleration with an application to the label switching problem

Steere, Edward

2016

http://hdl.handle.net/10539/22673

Steere, Edward (2016) Massive parallelism for combinatorial problems by hardware acceleration with an application to the label switching problem, University of the Witwatersrand, Johannesburg, <http://hdl.handle.net/10539/22673>

Downloaded from a DSpace repository. https://dspace.org/

Massive Parallelism for Combinatorial Problems by Hardware Acceleration with an Application to the Label Switching Problem

Edward Steere

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, in fulfilment of the requirements for the degree of Master of Science in Engineering.

I declare that this dissertation is my own unaided work. It is being submitted to the degree of Master's in Electrical engineering at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

.....
Edward Steere
Date: .....

Abstract

This dissertation proposes an approach to solving hard combinatorial problems in massively parallel architectures using parallel metaheuristics.

Combinatorial problems are common in many scientific fields. Scientific progress is constrained by the fact that, even using state of the art algorithms, solving hard combinatorial problems can take days or weeks. This is the case with the Label Switching Problem (LSP) in the field of Bioinformatics.

In this field, prior work to solve the LSP has resulted in the program CLUMPP (CLUster Matching and Permutation Program). CLUMPP focuses solely on the use of a sequential, classical heuristic, and has had success in smaller low complexity problems.

By contrast this dissertation proposes the Parallel Solvers model for the acceleration of hard combinatorial problems. This model draws on the commonalities evident in algorithms and strategies in metaheuristics.

After investigating the effectiveness of the mechanisms apparent in the Parallel Solvers model with regards to the LSP, the author developed DePermute, an algorithm which can be used to solve the LSP significantly faster. Results were generated from time based testing of simulated data, as well as data freely available on the Internet as part of various projects.

An investigation into the effectiveness of DePermute was carried out on a CPU (Central Processing Unit) based computer. The time based testing was carried out on a CPU based computer and on a Graphics Processing Unit (GPU) attached to a CPU host computer. The dissertation also proposes the design of an Field Programmable Gate Arrays (FGPA) based implementation of DePermute.

Using Parallel Solvers, in the DePermute algorithm, the time taken for population group sizes, K, ranging from K = 5 to 20 was improved by up to two orders of magnitude using the GPU implementation and aggressive settings for CLUMPP. The CPU implementation, while slower than the GPU implementation still outperforms CLUMPP, using aggressive settings, marginally and usually with better quality. In addition it outperforms CLUMPP by at least an order of magnitude when CLUMPP is set to use higher quality settings.

Combinatorial problems can be very difficult. Parallel Solvers has been effective in the field of Bioinformatics in solving the LSP. This dissertation proposes that it might assist in the reasoning and design of algorithms in other fields.

Acknowledgements

I would like to acknowledge the help and guidance I received from my supervisor, Professor Scott Hazelhurst, whose thorough approach to research challenged me to improve and whose involvement in the research community introduced me to good friends and colleagues. I would also like to thank the postgraduate students of the University of the Witwatersrand's school of Electrical and Information Engineering for two years spent learning to broaden my understanding of the world through discussion and exploration. I would like to acknowledge the role which my family provided in supporting me during the time I spent writing the dissertation and starting my career.

Contents

Abstract **i**

1 Introduction **1**

2 Background **5**
2.1 Combinatorial Problems . . . 5
2.1.1 Computational Complexity . . . 5
2.1.2 Combinatorial Problem Frameworks . . . 6
2.2 Algorithms . . . 6
2.2.1 Exact Algorithms . . . 7
2.2.2 Approximation Algorithms . . . 10
2.3 Metaheuristics . . . 11
2.3.1 Metaheuristic Models . . . 11
2.4 Practical Computation . . . 13
2.4.1 Parallel Model Classification . . . 14
2.4.2 Computer Architectures . . . 16
2.5 A Combinatorial Problem in Bioinformatics . . . 21
2.5.1 Population Structure . . . 21
2.5.2 The Problem Which the Model Based Approach Solves . . . 22
2.5.3 The Problem Which the Model Based Approach Creates . . . 23
2.5.4 The Label Switching Problem . . . 24

3 Parallel Metaheuristics and Solving the LSP **27**
3.1 Algorithms and Models . . . 27
3.1.1 Parallel Computation . . . 28
3.1.2 Computational Bottlenecks . . . 30
3.2 Practical Implementations . . . 31
3.2.1 Parallel Metaheuristics . . . 31
3.2.2 GPU Based Implementations . . . 32
3.2.3 FPGA Implementations . . . 35
3.3 The Current Approach to Solving the LSP . . . 36
3.3.1 A Greedy Algorithm Formulation . . . 36
3.3.2 A Fitness Function for the LSP . . . 37

4 The Parallel Solvers Model **39**
4.1 Introduction . . . 39
4.1.1 Trajectory-Based Metaheuristics Under a New Light . . . 39
4.1.2 Population-Based Metaheuristics Under a New Light . . . 40
4.1.3 A General Metaheuristic Model . . . 40
4.1.4 Class One Operators . . . 40
4.1.5 Class Two Operators . . . 40
4.1.6 A Universal Sequential Metaheuristic Model . . . 42
4.1.7 Information Sharing in Parallel Metaheuristics . . . 43
4.1.8 Parallel Strategies for Metaheuristics . . . 44
4.1.9 Geometric Division of Problems Solvable by Metaheuristics . . . 45
4.2 Specification of the Parallel Solvers Model . . . 45
4.2.1 Details of the Parallel Solvers Model . . . 46
4.2.2 A Universal Model for Parallel Metaheuristics . . . 46
4.3 The DePermute Algorithm . . . 49
4.3.1 Algorithmic Model . . . 50
4.3.2 Concrete Algorithmic Description of DePermute . . . 55
4.3.3 Data Structures . . . 65

5 Testing the Convergence Rates of Heuristics **73**
5.1 Motivation . . . 73
5.2 Test Design – Abstract Simulation . . . 74
5.2.1 Generation of Random Data Sets . . . 74
5.2.2 Population Diversification . . . 75
5.2.3 Mutation (Trajectory Diversification) . . . 75
5.2.4 Elitism (Population Refinement) . . . 77
5.2.5 Sub Space Division . . . 79
5.2.6 Communication . . . 81
5.2.7 Block Alignment . . . 83
5.2.8 Critical Discussion of Testing Procedure . . . 86

6 Practical Testing **87**
6.1 CPU Based Prototype . . . 87
6.1.1 Software Design . . . 87
6.1.2 Implementation of the DePermute Algorithm . . . 88
6.1.3 Data Structures . . . 93
6.1.4 Optimisations . . . 96
6.2 GPU Prototype . . . 97
6.2.1 Software Design . . . 97
6.2.2 Implementation of the DePermute Algorithm . . . 98
6.2.3 Data Structures . . . 103
6.2.4 Optimisations . . . 104
6.3 FPGA Hybrid Computer Prototype . . . 104
6.3.1 Software Design . . . 104
6.3.2 Implementation of the DePermute Algorithm . . . 105
6.3.3 Data Structures . . . 117

6.4 Test Objectives . . . 117
6.5 Time Based Testing Procedure . . . 117
6.5.1 Programs Under Test . . . 118
6.5.2 Test Platforms . . . 119
6.5.3 Program Compilation . . . 120
6.5.4 Data Sets . . . 121
6.5.5 Methods of Testing . . . 121
6.6 Random Data Set Generation . . . 121
6.7 Practical Data Set Acquisition . . . 122

7 Results **123**
7.1 Summary of Results . . . 123
7.2 System A . . . 127
7.2.1 Randomly Generated Data . . . 127
7.2.2 Real Data . . . 128
7.3 System B . . . 133
7.3.1 Randomly Generated Data . . . 133
7.3.2 Real Data . . . 133
7.4 System C . . . 135
7.4.1 Randomly Generated Data . . . 135
7.4.2 Real Data . . . 135

8 Conclusion **139**

A Extra Quality Results **141**
A.1 System B . . . 141
A.2 System C . . . 144

B Extra Sample Information **149**

C Scripts & Test Programs **151**
C.1 Random Data Generation . . . 151
C.2 Convergence Based Tests . . . 157
C.2.1 Shared Scripts . . . 157
C.2.2 Test 1 . . . 163
C.2.3 Test 6 . . . 166
C.3 Time Based Testing . . . 167
C.3.1 Random Data Set Testing . . . 173
C.3.2 Real Data Set Testing . . . 175

D Program Versions **179**

E Test Platforms **181**

F Extra Details of Algorithm Implementations **183**
F.1 Bitonic Sort 183
F.2 Parallel Prefix Sum 185

List of Figures

2.1 Allowed symbols per solution row *without mutation* given a starting population 13
2.2 Illustration of the relationship between the classes of parallelism in Flynn's Taxonomy 15
2.3 Layout of the GPU architecture 18
2.4 Layout of the Convey Hybrid Computer architecture 20
2.5 Example of the LSP problem for $R = 2$. Here the first two matrices are the original result, i.e.(O_0, O_0) and the second two matrices have been permuted to (O_1, O_1) 24

3.1 An illustration of the idioms of scientific computing as per Meswani et al. . . 30

4.1 Illustration of the parallel genetic algorithm using the parallel solvers model. 51
4.2 A general solution to the LSP in matrix format 54
4.3 Illustration of the process of finding block alignments 57
4.4 Illustration of the greedy cost algorithm (the depicted insertion is one of two needed to compute the lower bound) 60
4.5 Illustration of crossing two solutions with single point crossover. The first solution is selected randomly with a bias towards solutions with a high fitness and the second is selected at random. 63
4.6 Illustration of a $K = 4$, $R = 3$ LRT and its corresponding binary representation. Note that because all the nodes have been visited its binary representation has all bits unset. 69
4.7 On the left an un-searched LRT is depicted (note that all paths are available and all bits are set.) As partitions are made bits are unset and one edge in the path is fixed – eliminating 3 alternative paths. Two partitions of the LRT at this stage yield 16 LRTs. 70
4.8 The partial LRTs corresponding to the LRTs in the first partition of *Figure 4.7*. Note that instead of un-setting only one bit in the first row, we now set only the bits corresponding with the terminals of the edge which was traversed. . . 71
4.9 A sub search space in which two columns have been completed. The partial solution indicates the paths which have been taken while the partial LRT indicates those which may be taken next. 72

5.1 Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus Population Sizes for Various Problem Sizes 76

5.2 Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus Inverse Mutation Rates 78
5.3 Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus Elitist Proportions 80
5.4 Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus the Number of Divisions Made in the Search Space 82
5.5 Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus the Proportion of the Population Communicated 84
5.6 Graphs of the Number of Block Alignments Found For Various Problem Sizes 85

6.1 Module diagram for the CPU DePermute program 89
6.2 Outline of the DePermute algorithm for the CPU – part 1 of 2. 91
6.3 Outline of the DePermute algorithm for the CPU – part 2 of 2. 92
6.4 Modular decomposition of the GPU DePermute program 99
6.5 Illustration of the implementation of the DePermute algorithm for the GPU – part 1 of 2. 100
6.6 Illustration of the implementation of the DePermute algorithm for the GPU – part 2 of 2. 101
6.7 Modular decomposition of the FGPA implementation of the DePermute algorithm 106
6.8 Block diagram for the custom hardware implementation of the DePermute algorithm 108
6.9 State diagram for the Controller FSM 109
6.10 Block diagram of the memory controller sub blocks 112
6.11 State diagram for the Abstract Reader FSM 113
6.12 State diagram for the Abstract Writer FSM 114
6.13 Arrangement of FSM blocks in the Solution Loader block 115
6.14 Arrangement of FSM blocks in the Cache Loader block 116
6.15 Arrangement of FSM blocks in the Fitness Writer block 116

7.1 Graphical summary of the time based testing results achieved by CLUMPP and DePermute 124
7.2 Graphical summary of the quality of results achieved by CLUMPP and DePermute 125
7.3 Graph of *Time* versus (K) for randomly generated data testing of CLUMPP and DePermute run on system A 129
7.4 Graph of *Quality* versus (K) for randomly generated data testing of CPU and GPU DePermute run on system A 129
7.5 Graph of *Quality* versus (K) for randomly generated data testing of CLUMPP run on system A 130
7.6 Graph of *Time* versus (K) for real data testing of CLUMPP and DePermute run on system A 130
7.7 Graph of *Quality* versus (K) for real data testing of DePermute run on system A 132
7.8 Graph of *Quality* versus (K) for real data testing of CLUMPP run on system A 132
7.9 Graph of *Time* versus (K) for randomly generated data testing of CLUMPP and DePermute run on system B 133
7.10 Graph of *Time* versus (K) for real data testing of CLUMPP and DePermute run on system B 134

7.11 Graph of *Time* versus (K) for randomly generated data testing of CLUMPP and DePermute run on system C 136
7.12 Graph of *Time* versus (K) for real data testing of CLUMPP and DePermute run on system C 137

A.1 Graph of *Quality* versus (k) for randomly generated data testing of DePermute run on system B 141
A.2 Graph of *Quality* versus (k) for randomly generated data testing of CLUMPP run on system B 142
A.3 Graph of *Quality* versus (k) for real data testing of DePermute run on system B 143
A.4 Graph of *Quality* versus (k) for real data testing of CLUMPP run on system B 143
A.5 Graph of *Quality* versus (k) for randomly generated data testing of DePermute run on system C 144
A.6 Graph of *Quality* versus (k) for randomly generated data testing of CLUMPP run on system C 145
A.7 Graph of *Quality* versus (k) for real data testing of DePermute run on system C 147
A.8 Graph of *Quality* versus (k) for real data testing of CLUMPP run on system C 147

F.1 Bitonic sort network for eight nodes 184

Glossary

ADMIXTURE ADMIXTURE is a program used to estimate ancestry in unrelated individuals. 22, 23, 86, 91, 118, 119, 124

AMD Advanced Micro Devices is a company which designs and manufactures CPUs, GPUs and related hardware. 117, 192

AWS Amazon Web Services is a cloud-compute service provider which sells time on scalable on-demand hardware for various purposes. 192

CDF A Cumulative distribution function is a statistical means of illustrating the cumulative contribution of samples in different buckets. 51, 59, 91, 94, 97, 100

CISC Complex Instruction Set Computing is a name given to a family of processors whose instruction set is large and could contain instructions which combine many simpler instructions. 16

CLUMPP The CLuster Matching and Permutation Program is a program designed to search for optimal alignments of Q-Matrices from repeated runs of a population ancestry experiment. i, viii, ix, 3, 22, 25, 36, 85, 115, 119, 121–125, 127–129, 131–135, 137–140, 142, 143, 145, 146, 148, 149, 151–154, 156, 157, 184, 187

CPU A Central Processing Unit is a computer chip designed for general purpose program execution. i, viii, 3, 4, 19, 28–30, 33, 59, 85–90, 93, 95, 97, 100–102, 114, 115, 117, 121, 125, 128, 129, 134, 139, 145, 184, 187, 192, 193

CUDA Compute Unified Device Architecture is an NVIDIA branded technology designed to enable the use of NVIDIA cards in high performance computing. 117, 189

FGPA Field Programmable Gate Arrays are devices which are re-configurable in such a way as to enable the creation of arbitrary logic circuits. i, viii, 3–5, 19, 21, 28, 32, 33, 85, 95, 101–103, 106, 114, 145

FSM A Finite State Machine describes a machine which can occupy finitely many states and transitions between them which depend on inputs. viii, 106, 110–113

GCC The GNU Compiler Collection is a collection of open source compilers distributed by the Free Software Foundation. 94, 102, 117, 189

GPU The Graphics Processing Unit is a chip designed to excel at graphics related processing operations. i, vii, viii, 3–5, 17–19, 27–32, 42, 47, 85, 94–102, 114, 115, 121, 125, 128, 129, 139, 145, 184, 186, 193

GWAS A Genome Wide Association Study is an experiment designed to infer correlations between genetic data and a phenotype. 22

HDL Hardware Description Language is a programming language designed to support the description of hardware circuits for use in conjunction with programmable logic devices. 19, 33, 85

LRT The Leaf Root Tree is a specialised data structure devised as a means of efficiently describing solution spaces for the LSP. vii, 65–68, 70, 88, 92, 93

LSP The Label Switching Problem, as it relates to population structure studies, is to compute a consistent relabelling of columns in R Q-matrices such that each corresponding column in each Q-matrix has the same meaning. i, vii, 3, 4, 22, 24, 25, 34, 47, 48, 52, 53, 72

LUT A Look Up Table is an internal unit of an FPGA which enables the FPGA to be reconfigured. 19

MIMD Multiple Instruction, Multiple Data streams is a term used to describe computers which are capable of computing with multiple simultaneous instruction and data streams as input. 21

MIPS MIPS is a RISC architecture designed for CPUs. 16

NUMA Non Uniform Memory Architecture is a model for communicating between CPU PEs which, instead of sharing memory, uses message passing. 16

NVIDIA NVIDIA is a designer and manufacturer of graphics processing and related hardware. 17, 19, 28, 32, 116, 117, 189, 192

PCI Peripheral Component Interconnect is a specification for connecting hardware devices, usually to a motherboard, in a computer. 17

PE A Processing Element is any component of a computer which can be programmed to compute using instruction and data streams. 17, 27

PRAM A Parallel Random Access Machine is a generalisation of a sequential computer model known as the Random Access Machine which models computers which perform operations in lock-step over an arbitrarily large memory bank and which have arbitrarily many processing elements. 26, 27

PRNG A Pseudo Random Number Generator is a method of generating a sequence of numbers which appear to be random using an initial "seed" value (which is normally read from a source regarded as being random). 83, 94

RAM Random Access Memory is a volatile-memory chip designed for fast access of arbitrary memory locations. 26, 117, 192

RISC Reduced Instruction Set Computing is a name given to a family of processors whose instruction set is deliberately small. 16

RTL Register Transfer Language is an abstract methodology of programming programmable logic devices which is largely concerned with data's transfer and modification between registers. 21

SIMD Single Instruction stream, Multiple Data streams is a term used to describe computers which are capable of computing the same instruction, using multiple data streams as input, simultaneously. 17

SM A Streaming Multiprocessor, in GPU architectures, is a collection of PEs which all execute the same instruction. 17, 19, 97

SNP Single Nucleotide Polymorphism is a term used to describe variations in single bases of a genetic string which are known to vary between members of the same species. 22

TSP The Travelling Salesman Problem is a famous NP-Complete problem in which the shortest tour of all nodes in a graph must be computed. 9, 10

UMA Unified Memory Architecture is an model for communicating between CPU PEs which involves the sharing of memory to which PEs read and write. 16

Chapter 1

Introduction

Since its inception, the field of computing has undergone rates of change which are unparalleled in other scientific or industrial fields [12]. It was only in the 1930s that Gödel and Turing formalised the ideas of the limits of computability, leading ultimately to the formulation of an abstract computer (the Turing Machine). The Turing Machine is thought to be the first model for a computer which is capable of performing any computation that can be described by an algorithm. It catalysed arguments as to the limits of what can be computed, and as to the limits of the human mind [12].

During the 1940s digital computers became a practical reality. Since then the use of computers has evolved from the realms of pure thought and mathematics to (among others fields) the realms of business and social interaction. This has enabled the acceleration of large scientific computations in a period of time which would otherwise have been almost impossible. The field of computing is constantly evolving. This has lead to a myriad of demands on the capabilities of computing technology. Computer architects have historically responded by developing faster computers with ever smaller transistors. This enabled architects to include more transistors in more elaborate designs.

For decades this approach met the need for increasing speed in terms of the trend described by Moore's Law. Recently, this came to an abrupt halt due to a limiting feature which has come to be known as the "power wall." [2] Since the advent of the power wall computer architects have been forced to adopt parallel technologies to accelerate computer hardware. However, reliance on historic technologies and methodologies has not prepared the field for the relevant shift in paradigm which is needed in order to take advantage of parallel computing technology. A unique opportunity now exists to re-imagine the paradigm of computation and researchers are working hard to try and discover effective ways of addressing these changes [2]. The research in this dissertation was catalysed by this fundamental problem and how it affects the solving of combinatorial problems.

This dissertation is concerned with massively parallel computation and its application to combinatorial problems. In particular this dissertation investigates the applicability of massively parallel computation to solving combinatorial problems. Massively parallel computation is a particularly interesting component of the research because it has previously been considered less effective in the context of solving combinatorial problems.

NP-Hard combinatorial problems present a particularly interesting set of challenges – in particular:

- There are no known polynomial time algorithms to compute the optimal solution to many combinatorial problems.
- NP-Hard combinatorial problems occur often in nature and, due to their complexity, limit progress.

We are often forced to search for approximate solutions to NP-Hard combinatorial problems due to the limitations of modern computing hardware and the time frames we require for their computation. Of these algorithms the literature has established three categories of algorithm based on the class of heuristic which they employ. These are:

- classic heuristics;
- metaheuristics;
- hyperheuristics;

Algorithms based on classic heuristics have well established limitations which often lead to sub-optimal solutions. Meta and Hyperheuristics are both designed to address the shortcomings of classic heuristics. Metaheuristics can be considered master strategies for heuristics. Instead of constructing a solution iteratively, metaheuristics search through a space of constructed solutions. By contrast, hyperheuristics attempt to dynamically adapt the heuristic to a particular instance of a problem. This research focuses on metaheuristics for solving combinatorial problems.

Two models for metaheuristic algorithms are:

- trajectory-based metaheuristics;
- population-based metaheuristics;

We propose a generalised model for a parallel metaheuristic algorithm. The model relies on common attributes from both trajectory and population-based metaheuristics and offers a perspective which has previously been missed, i.e. metaheuristics stem from the same fundamental ideas and the classification of trajectory versus population-based metaheuristics hinges on the degree to which certain strategies are employed. The model is parameterised on these strategies and this research illustrates how the parameters can be adjusted to derive well known metaheuristic based algorithms.

Another important driving factor of this research is its application to the development of an algorithm to solve Label Switching Problem (LSP) quickly. This problem is found in population structure studies.

In a population structure study the researcher is interested in discovering the structure of genetic inheritance in a population. This can be established by comparing samples of the DNA which are known to vary among members of the same species. Using a model of these differences for an ancestral group we estimate a likely composition of percentage inheritance from a number of population groups for an individual with unknown ancestry.

Consider the scenario where a particular run of a population structure study establishes that a particular person is likely to have had inherited 40% of their alleles from one particular

ancestral group, 30% from another population group and 30% from a third group. However, due to the stochastic nature of the experiment we find that upon running the experiment a second time the values could have changed slightly. For instance, on the second run, we could find that the same individual is reported to have likely inherited 28%, 32% and 40% from the first, second and third groups respectively. Two problems arise with this data. The first problem is that the values change on subsequent runs of the experiment. It would make sense to take the average across the two runs. However, the second problem is that the order of the values has been shuffled in a seemingly random fashion, which prohibits us from doing so.

The random shuffling is a result of the program which is used in the experiment not being able to maintain a consistent ordering for these quantities across multiple runs. We call this shuffling "relabelling." This gives rise to the Label Switching Problem (LSP) – the problem being how to compute a consistent relabelling of the columns so that the quantities refer to the same population groups across multiple runs of the program.

The current state of the art software is a program called CLUMPP [24]. CLUMPP uses a classic heuristic based algorithm to solve the LSP. As a result the software suffers from the same problems as many algorithms based on classic heuristics – it often finds a sub-optimal solution.

An additional complication arises due to the complexity of the problem itself. The LSP is a computationally complex problem and it can take a long time to compute a solution of satisfactory quality even using an approximation algorithm such as that which CLUMPP employs. The field of population structure studies would benefit from a faster program to solve the label switching problem.

To that end I present DePermute. DePermute is an algorithm based on our generalised model of metaheuristic algorithms and is designed to be amenable to acceleration by massive parallelism. Three implementations of DePermute are presented for a CPU, a GPU and an FGPA hybrid based computer. We present the results of testing the CPU and GPU based implementations on randomly generated data sets and on a data set extracted from various population structure study project's data.

The structure of this dissertation is as follows.

In *Chapter 2* introductory material is presented. The chapter provides a grounding for the reader who is not familiar with important concepts from computer science via a short introduction to the algorithms and hardware which are used in subsequent sections of the dissertation. The chapter begins by formalising the concept of a combinatorial problem as it is used in this dissertation and then describes how any combinatorial problem can be solved given an adequate problem description and a method for generating every possible solution to the problem.

The chapter then describes how simple approaches to solving the problem fall short and the mechanisms which approximation algorithms use in an attempt to overcome these problems. In particular the chapter explores the difference between algorithms which provide a guarantee of finding the global optimum and the three classes of heuristic based algorithms which do not make the same guarantee.

Chapter 2 concludes with an introduction to the Label Switching Problem. The reader is introduced to the concept of population structure studies and in particular a model driven approach to determining population structure. Finally, an example of one of the complications which leads to the Label Switching Problem is presented.

In *Chapter 3* the literature regarding the use of metaheuristics in the design of algorithms to solve combinatorial problems is surveyed. This research supports a central tenet of this

dissertation – that there exist commonalities between metaheuristics especially with regards to parallelisation which have not yet been identified.

Chapter 3 begins by identifying the classes of parallelism usually used to solve combinatorial problems using parallel computing hardware. We contrast the three classes of parallelism against the realities of parallel computation – the limits on speedup which may be achieved in practise and the bottlenecks which have been documented in many programs used in scientific computation.

Then a survey of the literature regarding the implementations of algorithms to solve combinatorial problems is presented. *Chapter 3* concludes by narrowing the scope of the survey to the label switching problem and how it is solved with the current state of the art software. We conclude that a faster application would greatly aid scientific progress in the field because it would allow for larger instances of the LSP to be solved in a reasonable amount of time.

Chapter 4 serves two purposes.

1. it presents the author's work on amalgamating the commonalities between metaheuristic algorithms into a model the author has named the "Parallel Solvers" model;

2. it presents the DePermute algorithm – which the author has developed (based on the parallel solvers model) to solve the LSP;

Chapter 5 continues by testing the soundness of heuristics developed for the DePermute algorithm in *Chapter 4*. This chapter presents the premise that the efficacy of a heuristic can be determined by measuring its effect on the algorithm's rate of convergence on a solution of suitable quality without consideration of the time taken for iterations.

In *Chapter 6* and *Chapter 7* the efficacy of the algorithm is determined in a practical setting.

Chapter 6 presents three implementations of the algorithm for the purposes of testing and in order to establish whether the algorithm is effective under practical testing circumstances:

- DePermute is implemented on a:
  - CPU based computer;
  - GPU based computer;
- The design of an FGPA based hybrid computer implementation is also provided.

Chapter 6 presents a testing procedure which has been designed to determine how effective the implementations are when presented with randomly generated data and data from a typical data set. Following which *Chapter 7* presents the results of testing.

Chapter 8 concludes the dissertation.

Chapter 2

Background

This chapter provides two things to the reader. Firstly, a brief technical introduction to the reader who may not be acquainted with the fundamentals of computer science which the rest of the document depends on.

If the reader is unfamiliar with the GPU or FGPA device architectures then the author recommends examining the final section of this chapter for a primer on those topics.

Secondly, this chapter introduces the Label Switching Problem (LSP) and population structure studies. This section is important throughout the document because it describes the problem which is solved by three implementations of the new algorithm which this work presents.

2.1 Combinatorial Problems

2.1.1 Computational Complexity

Although the difficulty of any problem is intrinsic to the problem itself, an inadequate formulation of the problem can result in ineffective, long running algorithms. For instance it is possible to solve an easy problem using a formulation which makes the problem seem difficult. To illustrate this fact consider the following:

Example 2.1.1

The sorting problem.

Given a list of numbers in arbitrary order, produce an ordered list of those numbers.

A trivial solution to the problem could be to inspect every possible permutation of the numbers and accept the ordering which is sorted. Since there are $n!$ distinct permutations of n distinct numbers one could be lead to the belief that the sorting problem has complexity $O(n!)$. However, this would only be a result of a poorly formulated approach to the problem.

In practice the number of trial solutions which need to be inspected before finding the optimal solution is likely to be less than $n!$ because there is only one way to arrange the

numbers such that the last rearrangement is the last trial. Accordingly there is a $\frac{1}{n!}$ chance of the last ordering inspected being correct and (trivially) a $1 - \frac{1}{n!}$ chance of the solution being encountered before the last arrangement.

We are not certain whether many problems which appear to be difficult may be solved in polynomial time under an alternate formulation. There are still open questions about the complexity of various problems. It is in fact proven that if a polynomial time transformation of a problem in the complexity class *NP-Complete* to a polynomial time problem exists, then all problems in *NP-Complete* may be solved in polynomial time.

Importantly there exists no known algorithm to find the optimal solution to any *NP-Hard* problem in polynomial time. Therefore *NP-Hard* problems require that all solutions are inspected in order to guarantee that a solution is globally optimal.

"P is equal to NP" is one of the greatest problems facing the field of computer science and has inspired a great deal of research. The question remains unanswered. This work asserts that the computation of solutions to difficult problems in a realistic time frame requires a compromise.

For convenience, and in subsequent sections, NP-Hard combinatorial problems are referred to simply as combinatorial problems.

2.1.2 Combinatorial Problem Frameworks

In order to compute the optimal solution for any combinatorial problem a definition of a solution, a means of evaluating a solution's merit and an objective (minimisation or maximisation) is required.

We define a solution as a description of valid arrangements of symbols (the solution structure) where valid arrangements exist under a given set of rules (the solution rules) and the solution definition is denoted the letter S.

We denote the means of evaluating the merit of a solution F and note that semantically F behaves as a function mapping all solutions to numbers in $\mathbb{R}$ (the real numbers). If one enumerates all solutions to a problem and selects the solution for which the merit (evaluated by F) is closest to a desired value then one will have found the optimal solution. This algorithm is often referred to as the brute force algorithm because it does not use elegant means of reducing the problem complexity.

When formulating algorithms to solve combinatorial problems, it is useful to view the set of solutions as forming a search space which can be examined to discover the optimal solution.

The search space model makes it easier for search algorithms to be specified and reasoned about in human terms. For example one can visualise the well known Branch and Bound algorithm with a heat map representing the search space. The algorithm can be thought to be erasing cold parts of the search space if we determine that those parts will not contain the optimal solution[1].

2.2 Algorithms

Two broad categories of algorithm are used to solve combinatorial problems:

1. those which are guaranteed to find the globally optimal solution (exact algorithms);

[1] A process known as bounding the search space

Listing 2.1: The universal algorithm to solve any combinatorial problem

```
let start be an arbitrary solution under S

best  <- start
bestF <- F (start)
x     <- G (start)

while x != start:
    f <- F (x)
    if f < bestF:
        bestF <- f
        best  <- x
    x <- G (x)
```

2. those which attempt to find the globally optimal solution (approximation algorithms);

The following sections illustrate that exact algorithms have limited application in combinatorial optimisation because in the worst case their running time is prohibitively large even when intelligent means are used to reduce the running time of these algorithms.

2.2.1 Exact Algorithms

The simplest exact algorithm used to solve a combinatorial problem is the brute force algorithm. This algorithm is sometimes appropriate for small problem sizes because the overheads and complexity of implementing more elaborate algorithms might outweigh the simplicity of the brute force algorithm.

We highlight two popular alternative algorithmic models for specifying exact algorithms.

Branch and Bound

The advantage of a branch and bound algorithm is that it limits the bounds of the search space. If large portions of the search space are bounded then branch and bound is likely to inspect fewer solutions than the brute force algorithm.

In the worst case, branch and bound will still evaluate every solution. This can occur if the solutions differ by a very small margin. In these situations, the overhead of computing partitions and evaluating bounds results in a longer execution time

Branch and bound computes at most $O(n)$ divisions of the search space, where n is the number of solutions in the initial search space. For each iteration branch and bound must also compute h for each partition in the list. If the asymptotic complexity of h is defined to be constant (say, C) and the asymptotic complexity of F is defined as K then we have a worst case complexity for branch and bound of $O(Cn + Kn)$.

Dynamic Programming

Dynamic programming uses caching to avoid computation.

Algorithm

Begin by defining three entities:

- a search space S;
- an n-way partition function (which produces n distinct partitions of a given search space);
- a function h which produces a lower and upper bound of the effectiveness of a partition;

Initialise a singleton queue containing the original search space. The algorithm proceeds by dequeuing the next element in the queue and partitioning that element n-ways. All n elements are added to the queue after h is calculated for each sub partition.

In addition the algorithm keeps track of the lowest upper bound estimate of all partitions. If any element to be enqueued has a lower bound greater than the highest upper bound estimate, then that partition cannot contain a better solution than those elements already in the queue. We then remove the partition from the search.

By continuing this process the queue will eventually contain only one partition – the optimal solution.

Listing 2.2: Pseudo code to compute the *nth* Fibonacci number

```
let cache be an arbitrarily sized array with every element initialised to 0

fib n:
    if n == 0 or n == 1
        1
    else
        if cache[n] == 0
            cache[n] = fib (n − 1) + fib (n − 2)
        cache[n]
```

Consider the *nth Fibonacci number problem:* Given a number n compute the number which appears nth in the sequence of Fibonnacci numbers. A Fibonnacci number may be defined recursively as given in *Equation 2.3*.

$$fib0 = 1 \tag{2.1}$$
$$fib1 = 1 \tag{2.2}$$
$$fibn = fib(n-1) + fib(n-2) \tag{2.3}$$

This formulation requires exponentially many calls of the fib function before it will terminate, because for each evaluation of fib the lower terms are re-evaluated several times. Thus, for larger n the program will have excessive running time.

It is more efficient to use the arguments to the function as the key in a symbol table. The table thus acts as a mapping of the arguments of a function to its result, and provided that the arguments have been entered into the table beforehand, the problem of recomputing the body of a function is avoided.

Listing 2.2 illustrates an implementation of the fib function using dynamic programming.

A dynamic programming formulation of an algorithm can be used to solve the well known Travelling Salesman Problem (TSP).

Example 2.2.1

The Travelling Salesman Problem:

Given a graph representing a collection of cities, and the available roads between them compute the optimal ordering of cities so as to minimise the round trip distance, where no city may be visited twice[2] [17].

There are $n!$ many ways of arranging the n cities if there are connections between all cities. The performance of the brute force algorithm to solve the TSP is $n!$.

Held et al. present a recursive formulation of the TSP where the starting position is fixed as follows [22].

[2]The second constraint is not necessary, but avoids one having to consider the problem of negative cycles

Let $C(S, l)$ be the cost of starting at one city, visiting all cities in the set of cities S and then terminating at l and a_{xy} be the cost of a path between cities x and y. A piecewise definition for C is defined in *Equation 2.5*.

$$C(i, i) = a_{1i} \qquad \text{(i = 1)} \qquad (2.4)$$

$$C(S, i) = \arg\min_{s-i}(C(S - i, m) + a_{mi}) \qquad \text{(otherwise)} \qquad (2.5)$$

This recursive formulation enables dynamic programming to be used to formulate a more time efficient algorithm to solve the TSP. However the reduction in running time is not adequate for large problems. It has been shown that an algorithm to solve the TSP formulated using dynamic programming will execute in time proportional to $O(n^2 2^{n-3})$ and will require $\Omega(n2^{n-1})$ units of storage [22].

Similarly the running time of many exact algorithms is excessive when handling larger problem sizes. An alternative approach is needed to handle larger problem sizes in practical time constraints.

2.2.2 Approximation Algorithms

The reliable, fast, approximating computation of solutions to difficult combinatorial problems relies on the use of heuristics. A heuristic can be thought of as a rule of thumb which guides the way in which we explore a search space, or construct a solution, so that we are more likely to arrive at the optimal solution sooner.

For instance when addressing the TSP, a person could make the reasonable assumption that starting at one city arbitrarily and subsequently always taking the shortest path to any city that has not yet been visited (until all cities are visited) will result in a short tour. The selection of the city *closest to the current node* is termed a heuristic known as the Nearest Neighbour Joining heuristic.

This is an example of a greedy heuristic, so called because of the short-sighted nature of the algorithm's selection criteria. In the classification system presented by Crainic et al., the Nearest Neighbour heuristics are termed classical heuristics [13].

Algorithms designed using the Nearest Neighbour heuristic usually exhibit polynomial running time For example consider the Nearest Neighbour algorithm to solve the TSP. Beginning at one city arbitrarily the shortest path to another city is selected and that city is removed from the set S. This process continues until all cities have been visited.

In *Equation 2.7* the change is reflected by a minor modification to the recurrence relation for the exact solution. Note that one is no longer required to compute C multiple time for each decision made.

$$C(i, i) = a_{1i} \qquad \text{(i = 1)} \qquad (2.6)$$

$$C(S, i) = \arg\min_{a_{mi}}(C(S - i, m) + a_{mi}) \qquad \text{(otherwise)} \qquad (2.7)$$

The new complexity is therefore $O(n^2)$. To see this note that:

- At each call of C one must select the minimum from the set of distances to other cities in S.

- One must make this selection once for each city added[3].

It is noted in [13] that classical heuristics often result in sub-optimal solutions. This is because the process is halted immediately upon finding a local minimum, and because the methods used to counteract this effect (including random starting positions, multiple starting positions and combinations of classic heuristics) fail to satisfactory counteract becoming stuck.

2.3 Metaheuristics

Metaheuristics specify master strategies for searching neighbourhoods which contain better solutions according to set criteria.

Crainic et al. specify a six stage model for a metaheuristic:

1. *Initialisation* – selection of initial candidates and neighbourhoods;
2. *Neighbourhood Specification* – a specification of an area in the search space;
3. *Search Trajectory* – a mechanism for moving from one neighbourhood to another;
4. *Candidate Selection* – selection of solutions within a neighbourhood;
5. *Acceptance Criteria* – a measure of the effectiveness of a solution;
6. *Stopping Criteria* – the criteria to stop the search;

2.3.1 Metaheuristic Models

The intention of this research is to discover differences and commonalities in these algorithms. To this end the discussion turns to two metaheuristic models.

Simulated Annealing

Simulated annealing is a metaheuristic algorithmic model inspired by the similarities between the statistical mechanics of large multivariate systems and combinatorial optimisation, both of which are large multivariate discrete systems with many degrees of freedom [46].

Simulated annealing mimics the process of annealing whereby a material under study is melted and then cooled to discrete temperature levels and held for a time to allow for thermal equilibrium to be reached [46]. Annealing has enabled scientists to examine the behaviour of materials at low temperatures [46].

Algorithmically the process is simulated as a stochastic system [46]:

1. A random solution to the problem is generated.
2. The temperature of the system is initialised.
3. A random change to the solution is enacted.
4. If the change is positive it is accepted, otherwise:

[3] A more accurate bound can be established by $\sum_{i=1}^{n} i = 0.5(n^2+n)$ – a so called upper triangular comparison

Listing 2.3: A genetic algorithm

```
s     <- P random solutions in S
sNext <- []

while bestFitness > cutOff:
    f <- map F s                # Evaluate fitness
    for each index i in s:
        indiv1   <- randomSolutionBiased s f
        indiv2   <- randomSolutionBiased s f
        sNext[i] <- cross indiv1 indiv2
    bestFitness <- max f
```

A random number in $(0, 1)$ is generated.

If the random number is less than $e^{\frac{-\delta}{T}}$ then the change is accepted (where δ is the change in the objective function and T is the current temperature of the system).

If not then the change is rejected.

5. The temperature is reduced to the next level on the annealing schedule.
6. If the temperature is sufficiently low then the process is terminated.
7. If not then the process is repeated from point 3.

Genetic Algorithm

The genetic algorithm is a biologically inspired algorithmic model and is one of many so called evolutionary algorithms. Biological systems can be thought of as undergoing continual optimisation from one generation to the next, a process facilitated, in part, by the way genetic material is combined from parents during the formation of offspring.

If one assumes that those who are suited to an environment survive and produce offspring, then a combination of the genetic material of those who survive will comprise the genomes of offspring, and there is a chance that the random combination of two effective genomes will result in a more effective genome.

Algorithmically one requires at the very least, a reformulation of S, so that it can be thought of as a genetic string, as well as a method of evaluating the survive-ability (or fitness) of solutions (this is simply F). Using these two constructs a genetic algorithm may be devised as follows in *Listing 2.3*.

Genetic algorithms use various operations such as crossover and mutation. Crossover is the primary example of a genetic operator. It is the process whereby a child is added to the population by the random combination of its parents' genomes. One can incorporate additional genetic operators in order to more closely approximate the actual process in an attempt to improve the robustness of the process.

A practical genetic algorithm usually uses an operator known as mutation in order to improve the robustness of the search. The operator works as follows. For every offspring in the next generation, there is a chance that one of the symbols in the string will be randomly

mutated by randomly changing it to another valid symbol. Without the mutation operator, or a similar operator, genetic strings would be constrained as to what symbols are allowed to occur in each position by the symbols which occur in those positions in the initial population.

Figure 2.1 illustrates the symbols which can be found in each position in a genetic string if there is no mutation.

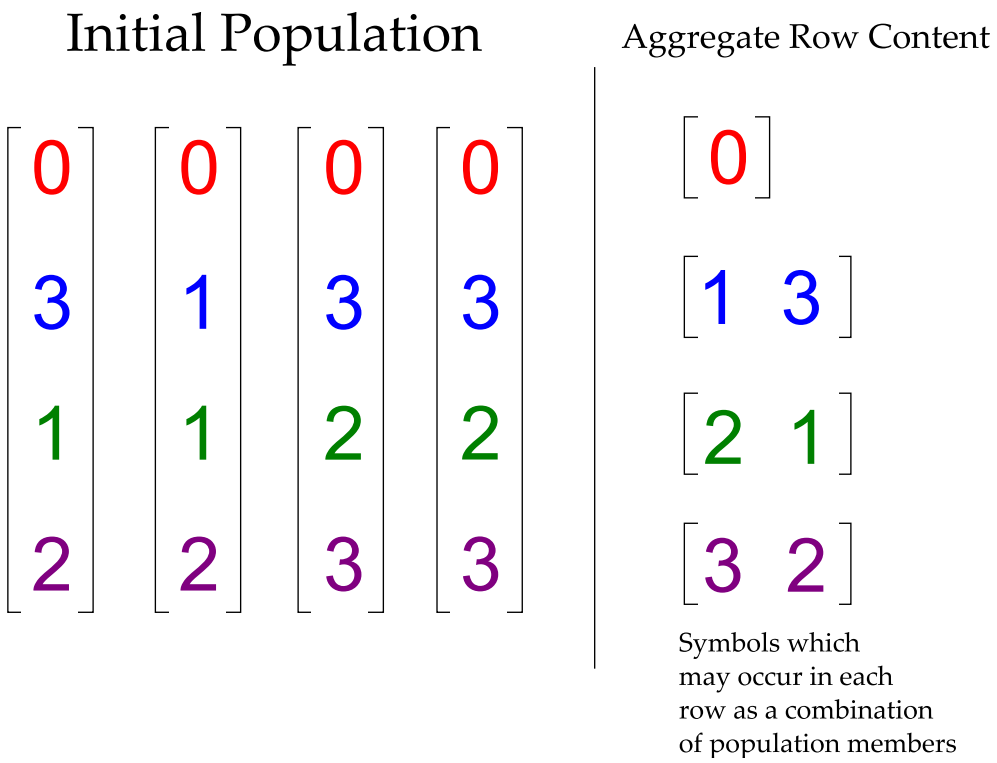


Figure 2.1: Allowed symbols per solution row *without mutation* given a starting population

2.4 Practical Computation

Progress in the field of combinatorial optimisation is measured as the ability to solve larger problem sizes than are currently possible.

Accelerating computation is primarily achieved by developing more powerful algorithms because algorithms define the amount of work which must be accomplished by the computer and this determines how long the computer must spend executing a task.

Alternatively, programmers look to more powerful computational hardware for progress. It is generally accepted that programmers used to wait until more powerful computational hardware became available with time as it was widely understood that programs would run faster on newer hardware. Recently, however, hardware engineers have begun to discover the physical limitations of the technology that had previously enjoyed incremental improvements. This limitation is known as the power wall and it puts an abrupt halt to the widely held assertion that software need not change in order to reap the benefits of increasing computational power [2].

It is now widely accepted that in order to achieve higher computational throughput, parallelism must be incorporated into processor design. The change has, however been somewhat superficial and computer architects have opted for the simple approach of replicating the core of the modern sequential processor to create multi-core machines.

In order to accelerate sequential processing these sequential processors require complex optimisations such as a large cache, pipelining and out of order execution. Such optimisations are expensive in terms of transistor real estate and can in some instances be comparable or larger in size than the functional units on the chips.

2.4.1 Parallel Model Classification

Flynn's Taxonomy is a useful model for classifying and reasoning about the style of execution adopted by various parallel computer architectures [52]. This model makes a high level distinction between execution models in two dimensions, namely:

1. whether the computer can process one or many instructions at a time;

2. whether the computer can process one or many units of data at the same time [52];

These two dimensions represent the two types of parallelism used in practical programming, namely pipelining (or task parallelism) and work division (or data parallelism) [52]. *Figure 2.2* illustrates the relationship between the classes of parallelism in Flynn's taxonomy [52].

- If a computer can process multiple instructions at a time, it is said to have Multiple Instruction streams (MI).

- If a computer can only process one instruction at a time it is said to have a Single Instruction stream (SI).

- If a computer can process multiple units of data at a time it is said to have Multiple Data streams (MD).

- If a computer can only process one unit of data at a time it is said to have a Single Data stream (SD).

Flynn's taxonomy does not accurately describe all parallel machines. For example consider the experimental systolic array computers of the pre 2000s [52]. In a systolic computer the inputs of Processing Elements (PEs) are connected to the outputs of other PEs and the PEs are arranged in a grid so that data propagates through the grid one unit per cycle until it reaches its terminus. When data has propagated through the entire grid it will have been processed.

Such a computer has both MI and MD streams. However the instructions in each unit are fixed and the model of a stream of instructions no longer conveys the reality [52].

Therefore Flynn's Taxonomy is complete for the class of all processors in which the notion of a stream of instructions and a stream of data is applicable.

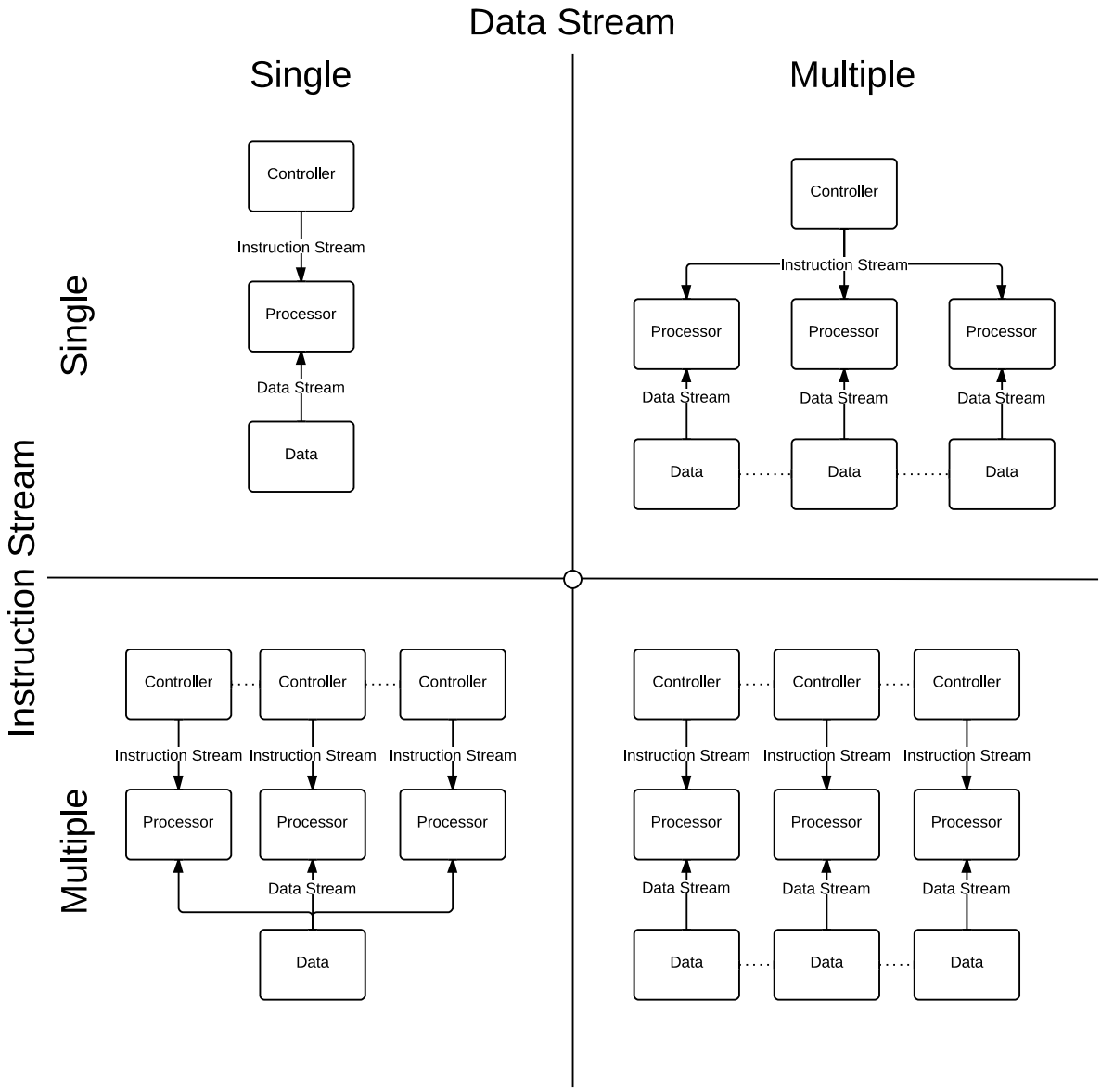


Figure 2.2: Illustration of the relationship between the classes of parallelism in Flynn's Taxonomy

2.4.2 Computer Architectures

CPU Architecture

The modern computer Central Processing Unit (CPU) has evolved through a long history of research, backward compatibility and cost constraints [45]. A brief history of these issues is presented [45].

Early computers of the 1960s and 1970s were limited by the high cost of memory, by the latency of fetching from memory and by the limits of the speed of local memory technologies. The Complex Instruction Set (CISC) architecture used by these early machines was an attempt to expose opportunities to combine many simpler instructions into one, so as to offset the cost of fetching the next instruction from main memory. (Instructions were stored in main memory because fast local memory was not yet in use.)

When the pitfalls of the CISC architecture became evident, there was a movement toward simpler instruction sets. This lead to both the MIPS and RISC architectures. These architectures were motivated by the fact that in a large instruction set, a small percentage of the instructions are ever in use. For example, in the CISC architecture of the early x86 processor, only ten of the instructions accounted for 95% of the SPECint92 benchmark [45].

Using a simpler set of instructions allowed the designers of RISC based computers to focus on effective optimisations – such as fewer instruction cycles per instruction decode, pipelining and hardware control. However, these architectures relied on optimising compilers and required a fast memory hierarchy.

Modern multi-core processors (which shall be referred to as commodity CPUs in subsequent sections) amalgamate many of the design principles, lessons learnt and legacy of their ancestors [44]. Modern processors are not based on RISC and have instead remained CISC because of legacy constraints. This has resulted in inefficiencies in transistor usage.

Multi-core commodity machines are MIMD. When each core is operating independently this architecture can be used to gain near linear speedup. However, most algorithms require some degree of cooperation between threads.

Cooperation is facilitated by communication, for which there are two major strategies:

- Uniform Memory Architecture (UMA) is a tightly coupled model for cooperating in which processors communicate by sharing memory [44].

- Non-Uniform Memory Architecture (NUMA) uses message passing and distinct images of memory to form a loosely coupled model.

In the design of computer architectures a significant challenge is how consistency is enforced between the memory for different PEs in the same processor. In a modern processor a completely consistent view is achieved through the use of snooping protocols [44]. Whilst these protocols simplify the task of writing correct programs, they are costly and make parallel programs with concurrent access to the same page unfeasible[32].

For these reasons modern processors often have a cluttered design [2], and the power barrier has afforded industry a unique opportunity to explore alternative computer designs and architectures.

GPU Architecture

The Graphics Processing Unit (GPU) is inspired by the execution patterns of modern graphical applications, including transforms, filters and other image processing algorithms.

Image processing algorithms often operate on a large number of units of data, usually in floating point format, arranged in matrix-like structures in memory. For example, the application of a blur filter to an image[4].

Two design aspects of modern GPUs have resulted from the bulk application of uniform operations to many units of data. Firstly, modern GPUs have a relaxed memory hierarchy allowing for the mutation of shared pages of memory without cache thrashing [23]. Secondly they comprise hundreds of simple units which all apply a simple instruction to different units of data [23].

GPU programming requires careful attention to how threads interact. An effective model for GPU programming is to use a basis in SIMD for understanding how the GPU executes instructions. There are three widely accepted principles of effective GPU programming:

- using an effective threading scheme which allows for threads to carry out as much work as possible independently;
- using a coalesced memory access pattern to bundle requests into single transactions;
- minimising communication, primarily between the host and the GPU, but also between the SMs and between SMs and higher levels of the memory hierarchy;

The use of these principles in an implementation is dependent on the algorithm. Therefore the level of success achieved in speeding up an application using a GPU is inextricably linked to the algorithm which is being implemented.

A GPU Architecture

The discussion will now turn to the NVIDIA Kepler GPU implementation of the GPU architecture.

A Kepler GPU consists of hundreds and even up to thousands of processing elements [42]. PEs are arranged into groups called Streaming Multiprocessors (SMs). Each SM contains a single instruction decoder. Therefore, each PE must execute the same instruction at any given time step. A Kepler GPU consists of fifteen SMs, a shared L2 cache, six memory controllers, a hardware thread handling engine, and a PCI express interface to a host computer [42].

Figure 2.3 illustrates the layout of the NVIDIA Kepler architecture.

GPUs are typically packaged as expansion cards which interface to a host computer. The host computer controls the execution of small performance critical pieces of code, suitable for the GPU, termed kernels [41]. When the host computer schedules new work to be carried out by the GPU, it specifies a configuration for thousands of threads, called a grid, to execute small units of work on the data.

Grids are multidimensional structures which allow threads to determine where their operation ought to be applied. Grids are allowed to contain more threads than can execute simultaneously on a GPU. The GPU therefore breaks grids down into sub-groups termed

[4] A blur filter is a simple operation which can be applied to each cell in a matrix representation of an image independently.

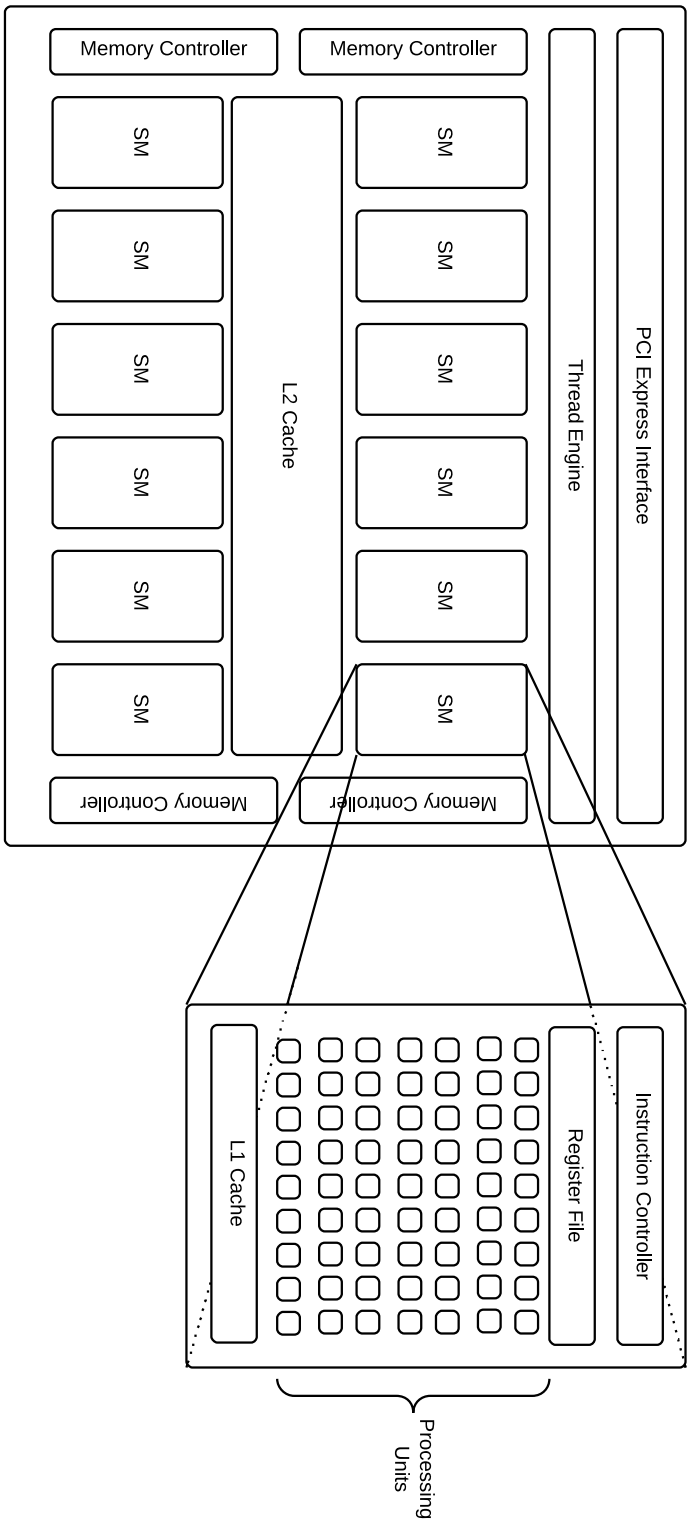


Figure 2.3: Layout of the GPU architecture

warps where the threads in each warp are executed in parallel, but the warps themselves are executed sequentially [41].

The grid programming model is very restrictive. It may seem that it does not allow for conditional expressions because of the lack of multiple instruction decoders per SM. However, NVIDIA and other GPU designers have relaxed this condition by allowing warps to be further broken into sub-groups which may executed different instructions should threads diverge [23]. This affords the programmer the flexibility of being able to specify some programs which require branching at the cost of context switching between groups of threads [23].

Field Programmable Gate Arrays (FPGAs)

The FGPA is a device which allows one to implement arbitrary digital logic circuits using a loadable description file. The file is initially programmed in an Hardware Description Language (HDL) and then translated to a machine readable file by a process which is similar to compilation. In this process code is interpreted and an optimised configuration of the FGPA is generated.

The core elements of an FGPA are lookup tables (LUTs) and programmable interconnects. An FGPA contains many LUTs arranged in a two dimensional array across the chip. In between the LUTs are programmable interconnects and on the outer edges of an FGPA chip are I/O blocks.

A LUT consists of an AND plane and an OR plane of logic [19]. Inputs to a LUT are arranged such that any input can be configured to connect to any one of the AND gates and the outputs of the AND gates are arranged such that any output may be configured to connect to any OR gate in the OR plane. A LUT may be used to describe any two level logic circuit.

The FGPA contains an array of LUTs with programmable connections between allowing the user to describe more complex digital circuits. The re-configurable nature of the FGPA results in overheads in time due to making interconnections between LUTs and between the planes in LUTs. In addition it is very difficult to determine the optimal translation of code and distribution of small units of logic to LUTs. Therefore FPGAs typically run at lower clock rates in order to accommodate for longer logic paths.

For this reason FPGAs are difficult to use. In addition the integration of an FGPA into a conventional computing environment often proves challenging and the overhead of communication between a host computer and an FGPA over say USB can outstrip any advantage initially envisaged in using an FGPA.

FPGA Hybrid Computer

The Convey Computer HC series computer is a “hybrid” computer [5]. It comprises a commodity computer with two CPU sockets and a co-processor board which houses four high performance FPGAs with their own memory subsystem. This enables the user to specify hardware to extend the architecture of the computer [5]. *Figure 2.4* illustrates the layout of the Convey Hybrid computer architecture.

The design philosophy is to integrate the extensions to the host architecture in a way which is opaque to the programmer [9]. The hybrid computer achieves this by providing supporting hardware. This is used as an adaptor between custom logic and the host machine

Figure 2.4: Layout of the Convey Hybrid Computer architecture

and custom compilers. These interleave instructions destined for the host computer and the co-processor in a single stream [9].

The programmer wishing to accelerate an application is advised to profile the application to determine which parts of the code constitute a computational bottleneck [9]. It is then suggested that the programmer use his understanding of the code to determine whether it would be amenable to an implementation in hardware. The design and implementation of suitable hardware is left to the programmer to undertake in Register Transfer Level (RTL) code [9].

RTL code uses a custom compilation process which integrates the user's hardware description with the standard interfaces to provided blocks. The resulting bit files are packaged into archives called "personalities" [9]. A personality can be swapped in at run time, using the Convey-provided libraries. This allows for integration of the personality with host binaries [9].

The co-processor board also houses eight memory controllers each capable of handling an instruction per clock cycle [10]. These controllers are optimised for random access in memory making them substantially faster at processing tree and graph like data structures than conventional DRAM memory controllers [10].

A program designed for the Convey HC computer is a combination of a commodity x86 program and instructions bound for application specific hardware. According to Flynn's Taxonomy, the FGPA hybrid computer falls under a potentially mixed or unclassified parallel class. However to the user, the system will appear to be MIMD because the host computer is still driven by a commodity processor.

2.5 A Combinatorial Problem in Bioinformatics

Up until this point the discussion has focused on providing a grounding in the algorithmic and architectural principles which are relevant to this dissertation. The discussion now turns toward a practical combinatorial problem which will be solved later in this dissertation. The problem is one of the central focuses of this dissertation and it originates from the field of bioinformatics.

Bioinformatics is a field of science which makes use of computer aided experimentation in genetic data. Computational methods have been used to infer the characteristics of a wide variety of biological systems. For example, Genome Wide Association Studies (GWAS) correlate genetic variation with phenotypical variation. The most commonly used variation for this type of study is known as a Single Nucleotide Polymorphism[5] (SNP) [16].

Many of these methods rely on statistical models of genetic markers [16]. The quantity, arrangement and presence of SNPs is used to compute the likelihood of their implication in phenotypes. This is done by comparing relevant statistics between individuals who do and do not present the phenotype [16].

2.5.1 Population Structure

Population structure is a term used in genetics. It refers to the hereditary composition of the individuals in a population. Population structure is a measure of the composition of ancestral groups which an individual was likely to have inherited genetic material from and the amount inherited from each group.

[5]Single bases distributed through a genome which are known to vary among members of the same species

Population structure studies are an important part of engineering GWAS studies among other things. If one is unaware of the underlying structure of a population then one may be mislead in making conclusions about whether a phenotype is associated with a particular genotype. For example, we may be interested in a particular disease and, in trying to discover its genotypical cause, sample a proportion of test subjects who are known to have the disease along with test subjects who are known to be healthy. From these samples we can use statistical testing to determine how likely it is that particular differences in the genomes of the two groups are culprits for the disease. However, the differences between the two groups may be the result of other effects.

Population structure introduces commonalities between the genomes of population groups. If a particular population has a prevalent occurrence of a particular SNP and all samples exhibiting the phenotype were taken from that group then we may be lead to believe that the SNP is implicated somehow in the manifestation of the phenotype we are studying. In actuality the SNP may or may not be the cause of the phenotype, but our analyses may be confounded by a lack of understanding its origin.

Understanding population structure is also important in formulating a better understanding of the history of species and migratory patterns throughout history. That information in itself is an important part of understanding the history of mankind and the biological context of nature.

Determining population structure isn't trivial. Many computer programs have been written to solve the problem. Various techniques and optimisations have been employed for improved performance of the software in some instances and in order to employ different approaches to estimation in others.

This dissertation will pay particular attention to the widely used and acknowledged program called ADMIXTURE. ADMIXTURE is a software tool which implements a model based algorithm to estimate the stratification of individuals in a population structure study [1]. Because of the stochastic nature of the ADMIXTURE algorithm (and other tools like it) it is common practice to average multiple result sets generated by the program. Averaging multiple ADMIXTURE result sets, although seemingly trivial, manifests a very difficult computational problem called the Label Switching Problem (LSP).

The state of the art program for solving the LSP is CLUMPP [24]. The CLUMPP algorithm employs a greedy heuristic which is fast, but in order to improve the quality of the search it conducts CLUMPP must run from multiple random starting points. This is a problem because, as Crainic et al. argue, multiple starting point classic heuristic algorithms are not immune to becoming trapped in local optima [13]. Furthermore even on aggressive settings CLUMPP can perform slowly.

2.5.2 The Problem Which the Model Based Approach Solves

One way of conducting a population structure study is to estimate which ancestral groups the alleles present in each individual originated from. For each individual we produce an estimate of the composition of proportions of genetic material which that individual is likely to have inherited from each of K ancestral groups.

In a population structure study one must either determine, or be supplied with, the number of ancestral groups from which the individuals could have inherited alleles. A discussion of the trade offs between the two approaches is now provided. Suffice to say that there are trade offs and that the model based approach has important applications and presents with

the LSP. We therefore consider the case in which the number of groups is supplied to the experiment.

Example 2.5.1

For a given individual $(0.2, 0.4, 0.4)$ means that the individual is likely to have 20% of his/her alleles inherited from population group 0, 40% of his/her alleles inherited from population group 1 and a 40% of his/her alleles inherited from population group 2.

The model adopted by ADMIXTURE postulates that there are K ancestral groups in a population where there are a number SNPs sampled from each individual [1]. The model hypothesises that each individual's genome (and therefore each individual's collection of SNPs) is formed by the random union of its parents SNPs. It then uses this assumption to formulate the problem in terms of maximum likelihood [1].

ADMIXTURE uses a block relaxation optimisation approach to resolve the likelihoods of individuals belonging to particular population groups [1]. This approach is stochastic and the results generated by ADMIXTURE vary.

2.5.3 The Problem Which the Model Based Approach Creates

A mean result can be established by running ADMIXTURE multiple times and calculating $\bar{x} = \frac{1}{n}\sum_{i=0}^{n-1} sample_i$. This resolves the problem of varying results in the output of ADMIXTURE. However, there is a more pressing problem which prevents one from simply averaging results. The problem is illustrated by way of example.

Consider the experiment where $K = 3$. A result set for a given individual could be $(0.01, 0.4, 0.59)$ in the first run of ADMIXTURE, $(0.43, 0.56, 0.01)$ in the second run and $(0.48, 0.48, 0.04)$ in the third run. The re-ordering of results in each of the three runs of the program does not mean that the underlying meaning of the results has changed. Each time the program was run it arbitrarily labelled the first quantity as column zero, the second as column one and the third as column two. The program assigns labels arbitrarily since the labelling doesn't impact the meaning of the results.

It is important to note that, not only are the results varied, but that the order of the numbers in each result set clearly correspond to a different ancestral population group in each run of the ADMIXTURE program. (How else could this individual be 1% population group zero in run one and then 43% population group zero in a subsequent run?) Unfortunately there is no way of forcing a consistent ordering of population group for each run of the ADMIXTURE program.

The ADMIXTURE program creates matrices in which each row corresponds to a result set for an individual. i.e. the first row corresponds to individual zero, and the second column denotes the proportion of his/her alleles which was inherited from population group one. Continuing on; the second row corresponds to individual one, the third to individual two and so on. This dissertation adopts the name "Q-matrix" for these matrices because this is the name which the ADMIXTURE program uses for them.

2.5.4 The Label Switching Problem

The Label Switching Problem (LSP) arises when the label corresponding to an experimentally determined quantity changes from one iteration of an experiment to the next.

Definition

The LSP:

Given K labels in a population structure experiment which is run R times, then the LSP is to compute an ordering $O = \{O_0, O_1, ..., O_{R-1}\}$ for each of R runs such that the n^{th} element in O_i shares a consistent label with the n^{th} element in O_j where $j, i \leftarrow 0 \dots R-1$ and $j \neq i$.

There are $K!$ candidates for the permutation of K labels which represents a consistent relabelling. We denote each of the K symbols $0 \dots K-1$, we denote a particular arrangement the letter O and finally label all possible permutations $O_0 \dots O_{K!-1}$. In this arrangement O_0 is the permutation $(0, 1 \dots k-1)$.

Note that we can select any permutation of symbols in each of the runs. For example, consider the problem where we have generated 4 result sets. If we decide to reorder the symbols in run 0 according to the permutation O_1, then the symbols for the first run will take on the second permutation, i.e. $1, 0, 1 \dots k-1$. Subsequently we reorder run 1 to permutation O_2, run 2 to permutation O_0; and run 3 to permutation O_1. This arrangement can be abbreviated to (O_1, O_2, O_0, O_1). (O_1, O_2, O_0, O_1) is distinct from (O_0, O_2, O_0, O_1) – note the difference in the first permutation. Note that one can keep run 0 fixed at O_0 ((O_0, O_1) is the same as (O_1, O_0)). Therefore there are $\prod_{i=1}^{R-1} K! = K!^{R-1}$ arrangements of the complete set of labels.

An example of two Q-matrices (which could have resulted from two runs of ADMIXTURE) is illustrated in *Figure 2.5*. In the example; the first two matrices are said to have permutation O_0 because the order of their columns has not been altered after having run ADMIXTURE. The second two matrices are permuted versions of the first two, shown in ordering O_1.

$$\begin{bmatrix} 0.2 & 0.5 & 0.3 \\ 0.7 & 0.2 & 0.1 \\ 0.3 & 0.2 & 0.5 \end{bmatrix} \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.2 & 0.1 & 0.7 \\ 0.2 & 0.5 & 0.3 \end{bmatrix}$$

(a)

$$\begin{bmatrix} 0.5 & 0.2 & 0.3 \\ 0.2 & 0.7 & 0.1 \\ 0.2 & 0.3 & 0.5 \end{bmatrix} \begin{bmatrix} 0.3 & 0.5 & 0.2 \\ 0.1 & 0.2 & 0.7 \\ 0.5 & 0.2 & 0.3 \end{bmatrix}$$

(b)

Figure 2.5: Example of the LSP problem for $R = 2$. Here the first two matrices are the original result, i.e.(O_0, O_0) and the second two matrices have been permuted to (O_1, O_1)

Jakobsson et al. propose that Q-matrices are likely to have a consistent labelling when the average similarity of columns which share symbols is maximised and propose that the Frobenius norm can be used to construct a suitable measure for similarity [24]. The Frobenius norm is defined as the square root of the sum of the euclidean distances between each corresponding column in two matrices [18].

With the objective function in mind one might restate the LSP in population structure studies as the computation of an ordering of columns for each Q-matrix which maximises the similarity of all pairs of columns in all Q-matrices.

In this chapter the problem has been introduced and a discussion of the fundamental theory of algorithms for solving combinatorial problems has been presented. The Label Switching Problem is an important combinatorial problem. The problem in itself is complex and a thorough treatment of it in an algorithmic sense is instructive from a theoretical perspective. The LSP in the context of population structure studies is also important in a practical sense. A faster program is required for solving the LSP and a fast program requires that an efficient algorithm be employed and that the program be designed for appropriate hardware.

Chapter 3

Parallel Metaheuristics and Solving the LSP

The implementation of programs to solve scientific problems often requires heroic effort in programming powerful computers to carry out calculations. The sciences, by their nature, are concerned with complex intellectual problems. Research into complex intellectual problems ought not to mandate that the researcher learns how to program super computers; therefore a collaborative effort, involving those invested in understanding computational problems and those who are invested in understanding nature, has gone into trying to understand how scientific programs can benefit from parallelism.

This chapter investigates parallel metaheuristic based algorithms and models in the context of available literature. It pays particular attention to implementations which have been undertaken on massively parallel computers. In undertaking this survey of the literature the chapter achieves two important goals:

- the discovery of successful approaches to parallelising metaheuristics for combinatorial problems;
- the identification of commonalities among these algorithms and approaches;

Finally, the chapter presents the literature relating to solving the Label Switching Problem (LSP). In particular it explains the approach adopted by the CLUMPP program, highlighting the computational characteristics of the algorithm employed by CLUMPP and the heuristics employed by the algorithm. This chapter is used in later chapters of this dissertation as the basis of an algorithm to solve the LSP.

3.1 Algorithms and Models

This dissertation is interested in a particular class of combinatorial problems which often require fast problem solvers in a practical setting. There are two important strategies which this dissertation employs. Firstly, parallelism and secondly the use of approximation algorithms.

Consider the parallelisation of metaheuristic algorithms. Crainic et al. highlight three parallelisation strategies which can be used to classify most parallel strategies for metaheuristics [13]. These classifications are:

1. "fine grained";

   The fine grained class of parallelisation includes strategies which seek to find parallelisation in the steps of the algorithm, either by analysing data dependencies and building pipelines or by finding opportunities for the parallel computation of small independent units of data [13].

2. "problem partitioning";

   The problem partitioning class of parallelisation describes algorithms which partition the problem arguments into smaller problems, computes sub-problems in parallel and then recombines the derived solutions of the sub-problems [13].

3. "cooperative parallelism";

   The cooperative parallelism class of parallelisation describes a strategy whereby multiple instances of the program are launched [13]. This class has been described as "multiple concurrent walks." Within this class two sub-classes exist: one for those algorithms which communicate and one for those which don't [13].

The three classifications are used to reason about prior work in the field.

3.1.1 Parallel Computation

First let us establish the boundaries of what is possible in terms of the acceleration of programs via parallelism. In this discussion an abstract model of a parallel computer is employed. The model is based on an abstract model of a sequential computer.

The Random Access Machine (RAM) is an abstract computer which can be used in the design of sequential algorithms. The RAM computer is defined as having a single processing element and an infinite memory in which every unit of memory can be accessed in a constant period of time. This model allows the designer to abstract themselves from the practicalities of hardware and to focus on the details of computation in an algorithm.

The generalisation of the RAM to a parallel machine is known as the Parallel Random Access Machine (PRAM) [26]. This machine extends the RAM model with infinitely many PEs all of which have constant time access to a shared memory bank [26]. The model also assumes that PEs operate in lock step[1] [26]. The model also encapsulates the semantics of concurrent operations on memory, with classifications for *exclusive* writing and *concurrent* writing. These can be used by the theoretician to build assumptions about the behaviour of operations.

A PRAM computer is capable of improving on the asymptotic running of particular algorithms, [26]; a short example is provided for convenience.

[1] Instructions are synchronised across all PEs

Example 3.1.1

The Sum Reduction Problem:

Given a list of numbers, produce the sum of the list of numbers.

The trivial solution to the problem is an $O(n)$ reduction of the list into a sum, through iteration, with an accumulator. Using a PRAM computer we can assign a PE to every odd index in the list. The numbers may be summed by selecting pairs in the list, summing them and then inserting those pairs into a new list. This process may be repeated until the list contains a single element. This element will be the sum of all other elements. This algorithm runs in time proportional to $O(\lg n)$.

How is the complexity of an algorithm altered by practical multi-processor machines? If the number of processing elements is constant, then note that parallelism can at best improve the running time of an algorithm by a scalar factor, that is, a constant number of PEs cannot affect the asymptotic complexity of an algorithm. Therefore, when the number of PEs is constant the asymptotic complexity of the parallel prefix sum algorithm is unchanged.

Show 1. *Consider a machine with p PEs. It can perform addition on at most $2p$ elements in parallel at a time. The number of additions which occur in parallel in the first pass is equal to $\frac{(\frac{n}{2})}{p}$. Subsequently, $\frac{(\frac{n}{4})}{p}$ additions can occur in parallel and subsequently half of those etc.* Expression 3.2 *formalises the resulting series.*

$$\sum_{i=2}^{\log_2 n} \frac{(\frac{n}{i})}{p} \tag{3.1}$$

$$= \frac{1}{p} \sum_{i=2}^{\log_2 n} \frac{n}{i} \tag{3.2}$$

As n increases in Expression 3.2 *the degree to which the expression is scaled by $\frac{1}{p}$ is unchanged. Therefore; the asymptotic running time of parallel reduction is unchanged if p is constant.*

This is an important result because it illustrates that the best which one can hope to achieve in a parallelisation is a faster running program and not an improvement in complexity. However, the PRAM has still been used as a means of designing effective parallel algorithms.

Dehne et al. used the PRAM as an abstract machine to design and test algorithms for the GPU [14]. They found that there were several non-trivial modifications required in order for the algorithms to perform satisfactorily on a GPU architecture. In addition to the well-established attributes of effective programs for GPUs (coalesced reads, minimal branching etc.) Dehne et al. noted that there are other differences between the PRAM and the GPU. Importantly, the PRAM model assumes that the PEs operate in lock step [14]. This is not the case on a GPU, therefore PRAM algorithms implemented on GPUs require the careful placement of thread synchronisation statements in order to maintain state invariants.

3.1.2 Computational Bottlenecks

Other researchers have made insightful discoveries when investigating the principles which are common to parallelism in computers. One such field of research is the automated discovery of bottlenecks in scientific applications and how best to leverage parallelism to achieve faster speeds.

Meswani et al. highlight that most bottlenecks in scientific applications can be described by one of six idioms [37]. *Table 3.1* lists those idioms along with pseudo code which describes the concept which they encapsulate.

Table 3.1: Table of idioms in scientific computing identified by Meswanin et al. [37]

Idiom Name	Pseudo Code
Transpose	$A[i][j] = A[j][i]$
Reduction	$sum+ = A[i]$
Stream	$A[i] = B[i]$
Stencil	$A[i] = B[i-1] + B[i+1]$
Gather	$A[i] = B[C[i]]$
Scatter	$A[C[i]] = B[i]$

The idioms listed in *Table 3.1* are illustrated in *Figure 3.1* for clarity. Note that the idioms (a-f) are listed in the same order as they appear in *Table 3.1*.

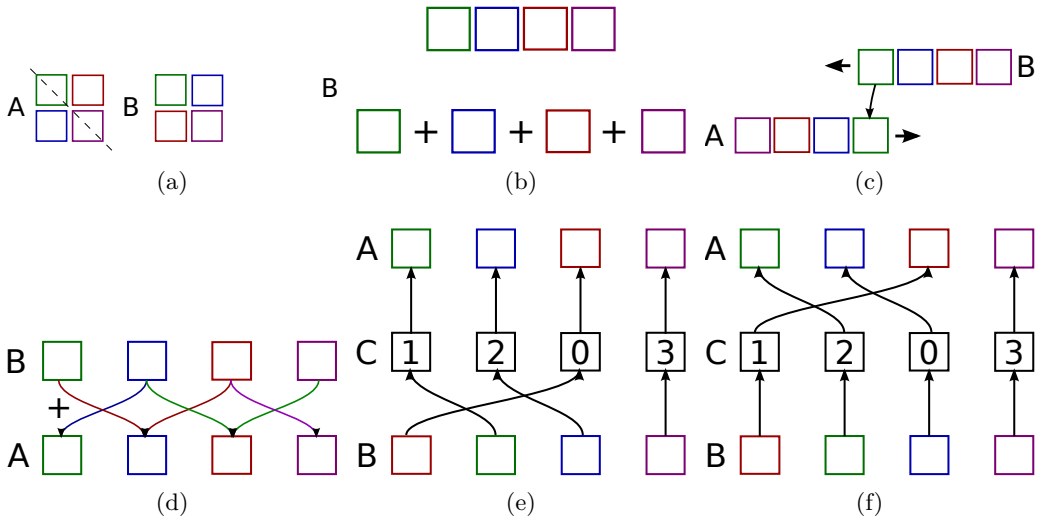


Figure 3.1: An illustration of the idioms of scientific computing as per Meswani et al.

Meswani et al. also present a series of tests to illustrate what memory bandwidth can

be achieved when executing each of the identified idioms on a commodity CPU system, an NVIDIA GPU and a Convey hybrid computer [37]. It was found that without taking into account the cost of transferring data to either of the co-processors, the co-processors both achieved higher memory bandwidth than the CPU for all idioms at most data sizes [37].

The GPU and FGPA hybrid computer achieved comparable results in most benchmarks, with the GPU outperforming the FGPA for Transpose and the FGPA outperforming the GPU for Scatter and Gather [37]. However, when the cost of transferring data, in terms of time, is taken into account both the GPU and the FGPA hybrid computer fail to outperform a CPU in a Scatter/Gather based benchmark [37]. Contrastingly, Carrington et al. adopted a more focused approach and paid specific attention to the Gather and Scatter idioms with the aim of developing a model as a means of:

- providing the programmer with knowledge of the whereabouts and presence of bottlenecks prior to searching [6];
- providing an estimate of the expected speedup that one can achieve when using a co-processor to accelerate computation for a particular program [6];

Carrington et al. were able to predict the performance of scientific applications accelerated using coprocessors using their model to within 10% [6]. In a case study it was found that an application designed using their model achieved speedups of up to 20% [6]. Although this result validates their model for approximating the speedup achieved using their model the result is very poor.

Carrington et al. ascribe the poor performance to the bottleneck in transferring data from the host computer to the coprocessor. The flaw in their methodology is that in simply migrating data to a device just in time for computation one creates significant overheads. Instead of directly mapping identifiable units of computation one might, for instance, restructure a computation such that there is less data transfer and the computation affords more opportunities for parallelisation. This alternative strategy might result in an algorithm which is less efficient on a CPU, but is more amenable to parallelisation on a GPU or other such device.

3.2 Practical Implementations

The discussion thus far has been general; it has elucidated commonalities between parallel metaheuristic algorithms and the principles of parallel computation. The discussion now turns to the implementations of parallel algorithms for combinatorial problems available in the literature and pays particular attention to those solved using massively parallel hardware. This section provides a basis upon which commonalities in approaches can be identified, thus allowing one to take better advantage of parallelism in computers.

3.2.1 Parallel Metaheuristics

Parallelism for metaheuristics has been researched since the early 1990s [33]. One early example is presented by Logar et al.; they propose a massively parallel implementation of a genetic algorithm using the MasPar computer. The authors were concerned primarily with the distribution of work and with how the semantics of the sequential genetic algorithm could be maintained in a massively parallel environment [33].

Logar et al. decided on what has become known as an "island" model to support work division in a genetic algorithm. Under this model a distinct run of the algorithm is performed on each PE. The semantics of the sequential algorithm are maintained by communicating sequences (members of the population) between PEs as a best attempt at maintaining a conglomerate population [33].

According to Crainic et al. the island model for a parallel genetic algorithm is a type three class (cooperative parallelism) of parallel metaheuristic, because it employs multiple cooperating runs [13]. The island model is the inspiration for many other genetic algorithms. In [26] the island model is used in the specification of a genetic algorithm for a Unix computer network and is extended to include the notion of trade agreements for optimal exchange of genetic material between islands [26].

Similar ideas are apparent in, and not limited to, the work of Jaros, Lazarove et al. and Borovska [3, 25, 31]. In most cases the island algorithmic model is successfully applied to achieve near linear speedups.

Borovska et al. present a more general solution to combinatorial problems in the form of a library for solving combinatorial problems using various parallel metaheuristics named PARMETAOPT [3]. A grid of suggested algorithms and parallel strategies is presented for a near complete listing of combinatorial problems [3]. The parallel strategies include: Master-Slave, Synchronous Iterations, Single Process Multiple Data, and Multiple Independent Runs [3]. PARMETAOPT conglomerates years of research into parallel metaheuristics for combinatorial problems. It can be considered a summary of best practices for solving combinatorial problems using parallel metaheuristics.

3.2.2 GPU Based Implementations

Interest in GPUs as a compute platform became widespread when vendors began supporting general compute programming for their devices. It has previously been commonly held that GPUs are ineffective at executing programs which involve irregular branching and irregular memory access patterns. This has lead to a commonly held idea – that GPUs are ineffective at running contemporary metaheuristic algorithms.

The research community has challenged the preconceptions of how effectively GPUs can accelerate metaheuristic based programs. In many instances this research has resulted in GPU based implementations which have outperformed their CPU based counterparts [14, 20, 23, 27]. A brief survey of research into GPU based implementations of metaheuristic algorithms is now presented.

Implementations of "Exact" Algorithms

Suri et al. design and implement a GPU algorithm for the exact solution to the multiple choice knapsack problem (a well known combinatorial problem) using a parallel dynamic programming algorithm [49]. Lalami et al.define the knapsack problem as follows in the next definition:

Although highly effective for smaller problems, dynamic programming doesn't scale well for large NP-Hard problem sizes in general because of large cache sizes and even larger constant factors. Another GPU implementation of an exact algorithm is presented by Lalami et al. in the form of a parallel branch and bound algorithm. However their algorithm is also not suitable for large problem sizes [30]. Given the failures of both algorithms to provide

Definition
Given items X_i, profit $p_i \in \mathbb{N}_+^*$ and weight $w_i \in \mathbb{N}_+^*$, where $i \in 1, \ldots, n$, and a knapsack with capacity $c \in \mathbb{N}_+^*$, find K_{max}, where:

$$K_{max} = \arg\max i \sum_{i=1}^{n} p_i x_i \tag{3.3}$$

$$\tag{3.4}$$

Such that K_{max} cannot exceed the capacity of the knapsack c. i.e.$\sum_{i=1} n < c$.

satisfactory scaling, it is apparent that an approximation algorithm is favourable for solving more general sizes of combinatorial problems.

Implementations of Population Based Metaheuristics

Population metaheuristic models are inherently parallel because the generation and evaluation of each individual in a population can be carried out in parallel.

Most work on GPU implementations of metaheuristics has primarily focused on so-called "population metaheuristics;" these are: Genetic Algorithms, Scatter Search and other biologically and sociologically inspired algorithms. The commonality in all of these heuristics is that they generate a collection of solutions on each iteration using the previous collection operated over by naturally inspired optimisation mechanisms. Therefore a population metaheuristic is any metaheuristic which actively computes and stores multiple solutions to a problem on each iteration.

Luong et al. propose that there are three models for parallel execution of the genetic algorithm [34]:

1. the parallel evaluation of the population;

   The first strategy is a class one (fine grained) parallel metaheuristic according to [13].

2. the parallel evaluation of solutions;

   The second is another example of a class one (fine grained) parallelisation.

3. the island model;

   The third is a class three (cooperative parallelism) parallelisation.

What follows is a (non-exhaustive) list of the implementations of population metaheuristic algorithms on GPUs available in the literature. The number and variety of implementations illustrate the diversity of approaches which are available from existing algorithms. A detailed description of each implementation is outside of the scope of this document. The examples are presented as a context for subsequent chapters.

Examples of type one parallelisation are found in the works of Zhu et al. and Krömer et al. [29, 55]. Zhu et al. use a parallel evolutionary strategy algorithm, which only differs from a genetic algorithm in that individuals possess collections of genetic information. Krömer

et al. use a simple Differential Evolution algorithm which naturally lends itself to the GPU compute environment.

The works of Jaros et al., and Mirsoleimani et al. illustrate the effectiveness of practical island based approaches to population metaheuristics for GPUs [25, 39].

Jaros et al. propose an implementation which uses multiple GPUs, one for each island, and assigns warps to collections of individuals. They cite various prior attempts which have used other assignment schemes limited either by the size of the individual or by the size of grids and or SMs on the device [25]. Jaros et al. reason that their approach is effective because it is scalable and it avoids thread divergence.

Mirsoleimani et al. propose a cell-based approach to a memetic algorithm [39]. A memetic algorithm combines a metaheuristic with a classic heuristic in a hybrid heuristic model. Mirsoleimani et al. combine a genetic algorithm and a variable neighbour search algorithm. Although the authors note that the so called cell model (in essence an island model with many small islands) for genetic algorithms would be effective for the GPU, the cell model doesn't integrate with the combined heuristic effectively [39].

Implementations of Trajectory-Based Metaheuristics

Metaheuristics which primarily alter the trajectory of a search through neighbourhoods have come to be referred to as trajectory-based heuristics. Melab et al. elucidate three models of parallelism applied to trajectory-based heuristics [36]:

1. multiple runs of the entire algorithm in parallel with no cooperation;

   a type three parallelisation (cooperative parallelism with no communication);

2. evaluation of solutions in a neighbourhood in parallel;

   a type one parallelisation (fine grained);

3. evaluation of a single solution in parallel;

   a type one parallelisation (fine grained);

Melab et al. extend a trajectory-based metaheuristics framework to allow for computation to occur on a GPU and reason that evaluating neighbourhoods in parallel is the most general model for parallelism in trajectory-based metaheuristics [36]. Melab et al. noticed that the amount of parallelism available in the evaluation of a single solution is entirely based on the problem being solved and is therefore clearly not general. Melab et al. don't use multiple parallel runs because that strategy serves to increase robustness (because we may select the best solution after many runs) and their goal is speedup [36].

Delévacq et al. propose a parallel model for Iterated Local Search (ILS) in which threads work on independent solutions but share members of a neighbourhood during neighbourhood evaluation [15]. The model was implemented on an NVIDIA Fermi GPU and achieved up to a six times speedup.

The speedup achieved by Delévacq et al. is disappointing, because it is complicated and difficult to implement an algorithm on a GPU and the GPU promises far greater peak performance figures. Unfortunately results similar to those achieved by Delévacq et al. are common amongst the articles surveyed.

Perhaps an alternative strategy for GPU based implementations of metaheuristics could be more successful. Perhaps the GPU is not amenable to the acceleration of trajectory-based metaheuristic algorithms. It is evident from this brief survey of the literature relating to GPUs in metaheuristics that the island model, communication and mapping of work to threads, SMs and grids are common motifs in the development of effective algorithms for implementation on a GPU.

3.2.3 FPGA Implementations

Field Programmable Gate Arrays (FPGAs) offer significant potential for hardware acceleration and could result in significant improvements in accelerating programs to solve combinatorial problems. However, FPGAs are notoriously difficult to program [50]. This difficulty is embodied by the fundamental shift in the way one must reason about the semantics of expressions when developing projects in a Hardware Description Language (HDL).

For example two statements in a computer programming language usually execute one after the other, however when implemented on an FGPA the two assignments will happen at precisely the same moment in time. These semantics allow the developer to make assumptions about synchronisation which aren't possible in a software development environment; however the semantics of HDL are also more complex and subtle.
There are three common strategies for accelerating computation using an FPGA:

1. avoiding overheads which exist in high level programming languages;
2. creating deep functional pipelines which overlap stages of computation – an instance of task parallelism;
3. replicating functional pipelines so that many units of data can be processed in parallel – an instance of data parallelism;

The Convey Computer HC-1 is an hybrid computer designed to allow the user to accelerate a program written in an high level language by offloading bottlenecks to a bank of FPGAs. Little research has been carried out on the Convey Computer in comparison to the GPU. This may be because software programmers are not suitably trained to use the machine and because it is less well known.

Uliana et al. argue that even though this platform has been developed with the explicit intent of making it more approachable to the developer it remains very difficult for the novice FGPA developer and they suggest that a visual approach using a block level design methodology may be more appropriate [50].

Previous work on the Convey computer includes that of Steinfadt et al. who use a dynamic programming formulation of the Smith Waterman algorithm to implement a systolic array on a Convey computer [48]. This is a successful technique because it is massively parallel and the simplicity of each cell in a systolic array lends itself to being replicated many times over.

In another instance Meyer et al. show how the Convey vector personality (instruction set) can be used in conjunction with the Convey OpenMP extensions to accelerate vector operations in normal applications [38]. Meyer et al. show that their strategy can result in a competitive speedup over an 8-core optimised version designed for a CPU [38]. In the opinion of Meyer et al., despite the fact that the framework allows for the application to be developed by a novice, the speedup achieved is insufficient to warrant the expense of the computer.

Walton et al. implement a parallel Scatter Search algorithm on a Xilinx Virtex 5 chip [51]. In the study a high level synthesis tool, called Handel-C, was used to compile programs written in the C programming language into HDL for synthesis [51]. In order to test both the efficacy of the FGPA based implementation and that of Handel-C all possible selections of the 12 optimisations which it offers as extensions to C were tested. The results reflect that the FGPA outperforms a CPU based implementation by up to 24×. However these results are somewhat dated because the CPU used in the test was a Core 2 Duo [51].

From the literature surveyed it is evident that what is gained in ease of programming with high level languages and commodity computers is lost when using highly specialised compute devices, and that the effort in programming specialised devices, whilst sometimes heroic, usually results in excellent performance figures.

It can be concluded from this survey, that the barrier to learning HDL and FGPA integration is a stumbling block for researchers.

3.3 The Current Approach to Solving the LSP

The number of solutions to the LSP is prohibitive. There are $K!^{R-1}$ solutions, therefore the brute force algorithm is only effective for small K and R and becomes impractical as the number of ancestral groups K and runs R increase.

A pragmatic approach to solving the LSP is to construct a solution by setting parts of the solution iteratively until a complete solution is constructed. This formulation – iteratively constructing a solution – is a classic heuristic. The general strategy for this heuristic is as follows. First arbitrarily select an ordering for the first run and then select orderings for subsequent runs which minimise the objective function for the partial solution.

This strategy significantly reduces the number of solutions explored, thus making the problem more computationally feasible. An algorithm which uses this strategy is said to be "greedy" because the algorithm applies the criterion of what would give the greatest immediate benefit, a short-sighted practise which tends to lead to sub-optimal solutions.

3.3.1 A Greedy Algorithm Formulation

Nordburg et al. published the first greedy algorithm for the LSP. It begins by selecting a Q-matrix (see *Section 2.5.3*) at random and shuffles it [40]. It then selects another matrix and computes the minimum cost permutation of that matrix to the first matrix and fixes it to that permutation. The algorithm continues to iteratively select matrices which haven't been fixed and computes the minimum cost in relation to the last fixed matrix [40].

Subsequently to Nordburg et al. the problem was investigated in greater depth by Jakobsson et al. [24]. Two greedy algorithms are proposed. These algorithms are implemented in the program CLUMPP.

Jakobsson et al. begin by defining a collection of orderings (the solution to the LSP) as a solution matrix, where the value at each index in each row corresponds to the index of a column, and where the symbol at the corresponding index in subsequent rows are to be interpreted as having the same label [24].

The first algorithm is called *Greedy*. Much like the algorithm in [40], the first algorithm does not permute each of R runs with respect to each other. Instead it begins by selecting a starting permutation $O_{initial}$ arbitrarily (usually at random) which constitutes the first row

a partial solution matrix. For $R-1$ iterations, the algorithm iteratively evaluates the cost of the partial solution with each of the $K!$ orderings for the next ordering added and selects the ordering for which the cost of the partial solution is minimised [24].

The second algorithm is called *Large-K Greedy*. It extends the concept of the first algorithm by selecting a random ordering for the first row of the matrix [24]. Then it constructs an ordering per run by iteratively adding the symbol to a partial row which minimises the cost of the column it is added to. This procedure is repeated iteratively until $R-1$ orderings have been constructed [24].

Greedy computes the objective function for $K!$ orderings per row thus computing the objective function (at least partially) $K!(R-1)$ times [24]. *Large-K Greedy* computes a symbol to add into each of the remaining positions in each row [24]. This is an upper triangular computation (like selection sort) thus resulting in $\frac{K(K-1)}{2}R$ (at least partial) fitness function evaluations [24].

3.3.2 A Fitness Function for the LSP

Jakobbsson et al. use the Frobenius norm to formulate a fitness function for computing greedy selections in both *Greedy* and *Large-K-Greedy* [24]. The Frobenius norm is defined in terms of the dimensions of Q-matrices in *Equation 3.5*, where C is the number of rows in the matrix (number of individuals in a population structure study) and K is the number of columns in the matrix (number of population groups in the study) [18].

$$||A-B||_F = \sqrt{\sum_{j=1}^{k}\sum_{i=1}^{C}(a_{ij}^2 + b_{ij}^2 - 2a_{ij}b_{ij})} \tag{3.5}$$

It is important to note that the selection of the minimisation of the distance between columns in result sets is a mathematical approximation of what may be correct in an ontological sense. One may perform further biological experiments, or use other means, to elucidate an understanding of what is correct biologically, but the minimisation of the Frobenius norm as a heuristic might not result in the same answer. Experimentation by heuristics allows one to gain useful insights into data and thus helps one to establish a better idea of what the ontological truth is.

Using the Frobenius norm Jakobsson et al. define two similarity measures, G and G', given in *Equations 3.6 & 3.7*, where W_K is a $K \times C$ matrix with all values set to $\frac{1}{K}$. *Equation 3.6* is designed to highlight instances where there is pronounced population structure in one of the matrices, i.e. when the values in a solution matrix are not evenly distributed per K entries in a row [24]. This formulation is effective because each row in a Q-matrix must sum to one[2].

$$G(A,B) = 1 - \frac{||A-B||_F}{||A-W_k||_F||B-W_k||_F} \tag{3.6}$$

Jakobsson et al. provide an alternate formulation in which similarity falls in $[0,1]$ using the normalisation constant $\sqrt{2C}$ [24].

$$G'(A,B) = 1 - \frac{||A-B||_F}{\sqrt{2C}} \tag{3.7}$$

[2]By definition one cannot have inherited more genetic material from the sample than was sampled.

The Frobenius norm has complexity $O(KC)$, from *Equation 3.5*. *Equation 3.6* has complexity $O(3KC) = O(KC)$ and *Equation 3.7* has complexity $O(KC)$, i.e. *Equation 3.6* has the same asymptotic complexity as its counterpart, but it has a lower constant factor. Jakobsson et al. show through empirical tests that the two similarity measures lead to different solutions using their algorithms, but that their properties are very similar [24].

Finally Jakobsson et al. define the fitness function as the average pairwise similarity between all matrices when permuted by the solution matrix [24]. The fitness function is defined in *Equation 3.8* (where $\{Q\}$ is the set of permuted output matrices) and has complexity $O(KCR(R-1))$ [24]. The factor $R(R-1)$ is as a result of the upper triangle comparison in *Equation 3.8*.

$$f(\{Q\}) = \frac{2}{R(R-1)} \sum_{i=0}^{R-1} \sum_{j=i+1}^{R} G(Q_i, Q_j) \tag{3.8}$$

As a measure to avoid becoming stuck in sub-optimal solutions, the user instructs CLUMPP to begin the search from a number of random starting positions. This is because a greedy search heuristic, starting from different partial solutions, will likely result in different solutions. The best of the resulting solutions is then selected.

Metaheuristic based algorithms are designed to solve the problem of becoming stuck. This dissertation asserts, as a central tenet, that metaheuristics are suitable for acceleration by massive parallelism, thus metaheuristic based algorithms are poised to address the limitations of the state of the art approach to solving the LSP.

Chapter 4

The Parallel Solvers Model

4.1 Introduction

This dissertation presents Parallel Solvers, an abstract model of computation which amalgamates the commonalities of parallel computation for metaheuristic-based algorithms. This chapter establishes a fundamental principle of Parallel Solvers and of this dissertation: that the trajectory and population classes of metaheuristic algorithm can be derived from the same algorithmic model. The relationship between the two lies in the amount of local (trajectory) versus collaborative (population) search which metaheuristic operations employ. It is also established that there are two primary methods of exploration which can be employed by local and collaborative metaheuristic algorithms:

- exploration by Diversification:

  Diversification is the process whereby solutions which are not currently under consideration are generated. This results in a more diverse body of solutions to be considered. Diversification acts to explore more of the search space. This mechanism of exploration attempts to avoid the problem of becoming stuck.

- exploration by Refinement:

  Refinement aims to improve on the best solution found. This process can be thought of as whittling down the body of solutions under consideration and is therefore destructive. Without a Refinement step a metaheuristic is simply a haphazard inspection of a search space in which we hope to find the optimal solution. In practice it is unlikely that a high quality solution can be found without some form of Refinement.

Diversification and refinement are established ideas from literature. This chapter proposes that the operators can be used to form the basis of a general model of metaheuristic algorithm and that they are therefore fundamentally important to the fields of metaheuristic algorithms and combinatorial optimisation.

4.1.1 Trajectory-Based Metaheuristics Under a New Light

The defining feature of a trajectory metaheuristic is that it considers a single solution. The solution moves through the search space on a trajectory which is modified by a local search

heuristic of some kind. The local searches at each point act as a diversification step because they introduce new solutions which were not originally considered. All solutions under evaluation will undergo some form of selection, which is usually weighted toward improvement in quality, and subsequently will update the current solution, thus refining the search.

4.1.2 Population-Based Metaheuristics Under a New Light

In a population-based metaheuristic there are two diversification processes. First a population of solutions is constructed by a stochastic process. The first process introduces a large number of solutions which were previously not under consideration by the algorithm. Secondly population-based metaheuristics often employ a process whereby solutions are randomly perturbed as the algorithm progresses.

Refinement in a population-based metaheuristic is implemented by operators such as selection, combination, preservation or elimination. Algorithm designers employ a combination of these operators in order to refine the average quality of solutions in the population. Intuitively the operators can be thought of as attempting to generate new solutions which combine the positive attributes of effective solutions.

4.1.3 A General Metaheuristic Model

We now examine and contrast the implementations of population-based metaheuristics to those of trajectory-based metaheuristics in terms of their counterparts in each class of algorithm.

There is no initial Diversification step in a trajectory-based metaheuristic and beyond the first step of the algorithm a trajectory-based metaheuristic doesn't store and operate on more than one solution across iterations. A trajectory-based metaheuristic may therefore be thought of as a population-based metaheuristic in which the population size is one.

A population-based metaheuristic randomly perturbs solutions during the search which it conducts (mutation is an example of a change which this class of algorithm makes). The process of randomly altering members of a population can be thought of as multiple distinct trajectory-based searches. For example, mutation in a population-based metaheuristic may be thought of as multiple small neighbourhood searches from each member of the population.

The mechanisms by which Diversification and Refinement occur in both population and trajectory-based metaheuristics are distinct. Two sub classes of Diversification and Refinement are introduced to address this distinction.

4.1.4 Class One Operators

I define the first class of operators as those which are derived from trajectory-based algorithms. *Listing 4.1* will be referred to as "trajectory Refinement."

The counterpart of trajectory Refinement is given in *Listing 4.2* and will be referred to as "trajectory Diversification."

4.1.5 Class Two Operators

Trajectory-based operators also have counterparts in class two population-based operators. *Listing 4.3* will be referred to as "population Diversification."

Listing 4.1: Trajectory Refinement Operator

```
// given:
// f(s)  A fitness function
// chi   A probability mass function
// s     The `current' solution in the search
// S     A set of solutions
// CDF   A function which produces the \gls{CDF} of a collection of solutions

cdf <- CDF(f, S)
rnd <- sampleRandomNumber (chi)
s   <- binarySearch (cdf, rnd)
```

Listing 4.2: Trajectory Diversification Operator

```
// given:
// chi  A probability mass function
// l    The number of solutions to sample
// s    A solution
// f    A function which produces a solution with a
//      random change from the given solution

neighbours <- [s]

for i <- 1..l
    neighbours <- neighbours + f(s, chi)
```

Listing 4.3: Population Diversification Operator

```
// given:
// P    The size of the initial population
// chi  A probability mass function
// f    A function which produces a solution at random

initialPopulation <- []

for i <- 1..P
    initialPopulation <- initialPopulation + f(chi)
```

Listing 4.4: Population Refinement Operator

```
// given:
// S        A set of solutions
// n        The number of solutions to generate
// chi      A probability mass function
// combine  A function which generates a new solution
//          via a combination of two solutions, or maintains a solution

nextGeneration <- []

for l <- 1..n
    nextGeneration <- nextGeneration + combine(S, chi, f)
```

Listing 4.4 will be referred to as "population Refinement."

4.1.6 A Universal Sequential Metaheuristic Model

Having established the fundamental means by which metaheuristic algorithms perform searches, the discussion now moves to the creation of a general model which encapsulates the principles of the two classes of metaheuristic algorithm.

Population-based operators are trivial in a population of one solution. The only rational definition of these mechanisms is the identity function, i.e. combination, elitism and other population-based mechanisms can be defined as identity for the unit population. Therefore, a trajectory-based metaheuristic can be thought of as a population-based metaheuristic with a unit population.

Using the definitions on class one and two operators from *Equations 4.1, 4.4, 4.2 and 4.2*, a universal algorithmic model for a metaheuristic is defined by combining these operators in *Listing 4.5*.

The model is general enough to encapsulate most of the ideas presented by the algorithms investigated in *Chapter 3*.

Example 4.1.1

Genetic Algorithm Example:

Consider the genetic algorithmic model. We can supply the universal algorithm with F as the uniformly distributed probability distribution function, g as the identity function and set l equal to one. $diversify_t$ along with $refine_t$ can be defined as a random perturbation to a solution. $refine_p$ can be defined as selection followed by combination. Finally $diversify_p$ can be defined as the random generation of a population of P solutions.

Example 4.1.2

Simulated Annealing Example:

Listing 4.5: The universal algorithmic model for a metaheuristic algorithm

```
given: S   // A solution definition
       P   // The size of the initial population
       l   // The number of solutions to sample in a neighbourhood
       F() // A probability mass function (potentially skewed)
       U() // A uniformly distributed probability mass function
       f() // A fitness function
       g() // Another fitness function

// Initial diversification
Solutions <- diversify_p (S, P, U)

// Iteration
while there exists no solution of acceptable fitness:
    Solutions <- refine_p (f, U, Solutions)
    for solution in Solutions:
        Neighbours <- diversify_t (S, U, l, solution)
        solution   <- refine_t    (g, F, Neighbours)
```

Consider the simulated annealing algorithm. The model nearly captures all ideas in simulated annealing. If P is set to 1, F is defined as the Boltzmann distribution and l is defined as the number of neighbours to consider then the resulting algorithm is very similar to simulated annealing.

Simulated annealing requires that the energy cost of evaluating neighbours is exported statically by the neighbourhood evaluation process so as to evaluate when thermal equilibrium is reached [46]. At the point where thermal equilibrium is reached F is altered by advancing the temperature to the next state in the annealing schedule.

The model doesn't support the principle of static data as such because there is no means of exporting static data and reading it at a later stage. The model could be extended to include a scratch space, but this is somewhat unsatisfactory because it breaks the mathematical purity of the model.

The pivotal difference in the formulation of both the Genetic and Simulated Annealing algorithms is that the population-based metaheuristic has $P > 1$ and the trajectory-based metaheuristic has $l > 1$. A trajectory-based metaheuristic algorithm is therefore the counterpart of a population-based metaheuristic in terms of the degree to which the algorithm maintains solutions across iterations versus inspecting neighbours.

We note that a practical (and interesting) algorithm might result by setting both P and l to be > 1.

4.1.7 Information Sharing in Parallel Metaheuristics

Previously, type three parallelisation (cooperative parallelism) was defined as a scheme in which concurrent instances of a metaheuristic are spawned which may or may not communi-

cate [13]. It was argued that cooperative parallelism allowed for a more complete search to be undertaken [13]. *Chapter 3* presents research which argues that cooperative parallelism is limited in its application because it is not suitable for a trajectory-based metaheuristic. We can infer from this argument that the researchers believe the sharing of information to be a detail with respect to implementation rather than algorithm.

The mechanism which sharing promotes is a convergence on what has been identified as a promising area of the search space. This requires that an entity is capable of communicating one or more promising solutions and of adapting its own search upon receiving information from other entities.

In a population-based metaheuristic, integration of this information is trivial. One can simply integrate solutions from one or many of the instances of the search into other instances. A trajectory-based metaheuristic presents a greater challenge. A proposed method is, as the receiver, to include communicated solutions into the next neighbourhood search and, as a communicator (which all instances are), to communicate a selection of solutions, which came under inspection at some point in the search, to the other solvers.

There are potential mechanisms for communication in both population and trajectory-based metaheuristics. This highlights the possibility of a more general treatment of information sharing in the context of parallel metaheuristic algorithms.

4.1.8 Parallel Strategies for Metaheuristics

The "island model" is a common motif in the specification of parallel strategies for population-based metaheuristics. The island model is a mechanism of sharing the information in metaheuristics and is mostly applied to population-based metaheuristics. Distinct populations are modelled as "islands" (concurrent instances of the search) and individuals can travel between islands by ships with limited passengers.

The literature indicates that the island model has been used to provide satisfactory speedup and improvements to solution quality [25, 39, 33, 26]. In cases where this model was implemented on a GPU, data parallelism was also leveraged within islands to accelerate the rate at which steps were carried out [33]. The use of data parallelism is a somewhat confounding factor, because it is not possible to determine whether the successes found in the literature were a result of the communication step in the island model or were a result of the improved rate at which iterations could be carried out.

The island model also bears a similarity to the granularity and the hierarchical nature of modern computers. Although originally used on coarse grained architectures (such as networked computers) the memory hierarchy of a modern computer, especially in a massively parallel computer, has similarities to the island model. On a high level a computer has slower memory and communication busses between processing elements – or in the case of the GPU – between SMs. Therefore, a coarsely grained division is suitable. On a low level, there is fast local memory (suitable for sharing data in the case of the GPU) which is suitable for tightly coupled finely grained operations.

An effective algorithm can lead to programs which execute a low number of iterations before converging on a suitable answer. This results in a shorter running time for the algorithm, an attribute we refer to as the speedup due to *convergence*. An effective algorithm can also lead to programs which execute iterations more quickly due to data parallelism, an attribute we will refer to as the speedup achievable by *parallelism*.

Convergence

If the number of iterations which a "single island" algorithm executes before finding a suitable solution is denoted as n, and the number of iterations which a "multi-island" algorithm makes before finding a suitable solution is denoted as N, then the algorithm can be said to have been sped-up by an amount proportional to $\frac{n}{N}$.

Parallelism

If a computation can be decomposed into discrete steps such that there is no data dependency between the steps in each unit, then one can perform each sub-computation in parallel. The speedup we can achieve in this instance is a factor of the number of processing units available and is limited by Amdahl's law – speedup achievable through parallelism is limited by the proportion of a program which must be sequential [32].

If a program can execute fewer iterations *and* each iteration is executed in a shorter amount of time, then we are guaranteed to achieve a speedup. Therefore a suitable combination of the two strategies is a sound strategy for parallelising an algorithm.

4.1.9 Geometric Division of Problems Solvable by Metaheuristics

Often the most convenient and effective method for developing a parallel algorithm is to simply divide the problem into partitions, and to compute partitions in parallel. This partitioning scheme is referred to as geometric partitioning in this dissertation because of the parallels it draws with geometric divisions in a mathematical sense. If a problem is amenable to this strategy it is often referred to as embarrassingly parallel.

A geometric partitioning scheme has not been investigated in the context of population-based metaheuristic algorithms. It is unclear whether a geometric partitioning scheme would result in a speedup in the processing time for a metaheuristic. Simply dividing the search space into partitions won't necessarily result in a speedup, because a metaheuristic search is non deterministic by nature. It might be the case that the algorithm is in fact more efficient when the search space has not been divided. Finally it might be the case that experiments designed to determine and contrast the efficiency of either approach aren't repeatable. This is because of the unpredictable way in which metaheuristics-based algorithms conduct searches.

My intuition is that a partitioning scheme would assist an island model algorithm to perform a more comprehensive search, because each island would be constrained to a particular division for a time, thus ensuring that areas are not immediately overlooked, but still allowing for islands to converge on promising areas of the search space.

4.2 Specification of the Parallel Solvers Model

The Parallel Solvers model is a parallel extension of the universal model for metaheuristic algorithms. The goal of the model is to incorporate effective strategies in parallel metaheuristics, thus being general enough to enable it to describe most parallel metaheuristic algorithms.

This section explores cooperation accelerated by low level parallelism in the hope of elucidating a robust model for the development of algorithms which are suitable for parallelism – especially in a massively parallel sense.

The remainder of this chapter is structured as follows:

- details regarding changes to the algorithm are presented;
- finally the model itself is presented;

4.2.1 Details of the Parallel Solvers Model

Parallel Solvers are modelled on a grid search with cooperation as an abstract heuristic. The work of an entity searching in its allotted region is modelled by a local search carried out by a metaheuristic based algorithm. The local search starts in such a way that none of the solutions it considers are outside of a partition in the search space. This can be facilitated by applying rules to the random generation of solutions during initial Diversification.

To avoid heavy overheads, as there may be when restricting Diversification, strict partition rules can be relaxed so that Diversification is allowed to escape a partition but still *tries* to stay in its scope (example to follow). Correct communications, guarantees on convergence and predictable behaviour are assured by way of a contract which a parallel solver must implement.

The contract can be described by 4 simple rules. Given a parameter n:

1. A solver must keep track of the n best solutions encountered thus far.
2. A solver must be able to communicate the best solutions it has kept track of.
3. A solver must be able to incorporate n solutions into its search.
4. A solver must produce solutions which are monotonically increasing in fitness.

The first three rules constitute a simple cooperative contract without specifying the topology of the communication channel. The fourth rule is a result of the first rule, i.e. if the solver is capable of tracking the best solutions found thus far then the solutions which it produces will naturally never decay in quality. This need not constrain the solver to local optima, because the best solutions encountered can be stored in a separate buffer which doesn't play a role in the execution of local searches.

4.2.2 A Universal Model for Parallel Metaheuristics

The discussion now moves to the extensions which Parallel Solvers make to the universal model. The first addition to the model is a communications channel. The channel is modelled as a stream so that the details of the implementation (synchronicity, topology etc.) are left to the designer.

Listing 4.6 defines an extended universal model – Parallel Solvers. C, n and i are new parameters, i.e. they were not present in the sequential model. C controls the number of solvers to use in the search. n defines the number of solutions to be communicated between solvers. i indicates the degree of trajectory (denoted by a subscript "t" in listings) or population-based (denoted by a subscript "p" in listings) search which is to be undertaken by the metaheuristic.

i defines the number of solutions which ought to be integrated to the neighbourhood search, e.g. for a pure population-based metaheuristic i should be set to 0, i.e. all communicated solutions are incorporated to the local population. For a pure trajectory-based metaheuristic

Listing 4.6: The algorithmic model of parallel solvers

```
given: S    // A solution definition
       P    // The size of the initial population
       l    // The number of solutions to sample in a neighbourhood
       F()  // A probability mass function (potentially skewed)
       U()  // A uniformly distributed probability mass function
       f()  // A fitness function
       g()  // A secondary fitness function
       C    // The number of solvers to spawn
       n    // The number of solutions to communicate
       i    // The number of solutions to integrate into the neighbourhood search
       comm // A communications stream

// Determine partitions of the search space
S_subs <- partition (S, C, f)

par:
    // Determine location in grid
    id    <- solverID

    // Determine initial search space
    S_sub <- S_subs[id]

    // Initial diversification
    Solutions <- diversify_p (S_sub, P, U)
    Best      <- {}

    // Iteration
    while there exists no solution of acceptable fitness:
        Solutions <- refine_p (f, U, Solutions)
        if terminate? comm
            comm << terminate
            terminate
        comm >> Communicated
        // Replace last Solutions with (n-i) communicated solutions from
    Communicated
        Solutions[-(n-i)::] <- Communicated[0:(n-i)]
        for solution in Solutions:
            // S_sub can be replaced with S to relax scope requirement
            Neighbours <- diversify_t (S_sub, U, l, solution)
            // Add the remaining communicated
            Neighbours <- Neighbours + Communicated[(n-i):n]
            solution   <- refine_t    (g, F, Neighbours)
        if updateBest? (Best, Solutions)
            best <- updateBest (Best, Solutions)
        comm << Best
    comm << terminate
```

i should be set to n so that all communicated solutions are incorporated to the neighbourhood search[1].

Notable programmatic additions to the model include:

- the addition of a partition step:

  The partition step is abstract allowing the designer to decide on an appropriate method for partitioning. A proposed method is to use a branch and bound based algorithm which halts when enough partitions have been created. In doing so, areas of the search space which contain low quality solutions could be avoided by removing them from the search early on.

- the par construct for parallel execution:

  Par is used in the model to represent a portion of the algorithm which can be executed with asynchronous concurrent semantics.

- communications constructs:

  The C++ style stream operator is used to represent the sending and receiving of data from the communications channel *comm*. *comm* >> *variable* represents a value being received from the channel and stored in a variable. *comm* << *value* represents a value being sent over the channel. The *comm* entity resolves topology, serialisation and any other requirements for communication opaquely so that the focus can remain on the algorithm itself.

- an extra termination condition:

  Finally an extra termination condition is added, that is, when any solver flags that the search should terminate then terminate this solver as well. This occurs when any solver finds a solution with sufficient quality, thus escaping and flagging terminate to the communications channel.

This section concludes by presenting two examples of how the Parallel Solvers model can be used to design a parallel genetic algorithm and a parallel simulated annealing algorithm.

Example 4.2.1

Genetic algorithm:

A parallel genetic algorithm can be specified using the Parallel Solvers model by setting P to a value greater than one. The parameters which are shared with the sequential model remain unchanged for the specification of the parallel counterpart. n can be set to a value less than P so that only a portion of the population is communicated. i is set to zero so that none of the received solutions are integrated into a neighbourhood search.

The generation of solutions within a sub-space can be time consuming. Therefore, the condition that solvers stay in their search space can be relaxed for the $diversify_t$

[1] Just as is the case in the sequential model, there exist opportunities to hybridise the use of class one and two operators.

step of the algorithm. This decision affords the designer of an algorithm with the choice of whether to allow the searchers to escape their division.

Example 4.2.2

Simulated annealing:

A parallel simulated annealing algorithm can be specified using the Parallel Solvers model by setting P to one. The temperature must again be statically exported (breaking from the model) and the parameters which are shared with the sequential model can remain the same.

i is set to n so that all communicated solutions are included in the neighbourhood search, i.e. when evaluating states the algorithm first takes the communicated solutions under consideration – instead of randomly generating every neighbour solution.

4.3 The DePermute Algorithm

DePermute is one of the major contributions of this dissertation. It is a parallel metaheuristic based on the parallel solvers model. DePermute implements a genetic algorithmic approach using parallel solvers. It emphasises the use of class two metaheuristic operators (population diversification and refinement). DePermute, generates an initial population at random and subsequently refines it using a combination of crossover, elitism and parallel cooperation. DePermute introduces diversification on a per iteration basis using the mutation and recombination operators.

One of the key principles of the algorithm is its focus on data parallelism – which it supports by way of class two operators. DePermute's focus on data parallelism introduces opportunities for a trivial division of work among PEs. The GPU model of computation supports this strategy well because it emphasises the application of operations across batches of data with little branching. By employing this strategy, DePermute trades memory efficiency for more opportunities to use data parallelism.

A potential pitfall of this approach is its reliance on initial diversification in generating sufficient entropy from which a solution is derived. Note that the word "entropy" is used in this dissertation to describe a property of chaotic systems which results in unpredictable behaviour. If insufficient entropy is present in the initial diversification step then the algorithm will never find the optimal solution. This point was illustrated in *Figure 2.1* and it was concluded that there is a permitted set of symbols per position in a "genetic sequence" given an initial population.

There are two mechanisms which can be used to combat this effect. The first of these is the generation of very large initial populations in an attempt to increase the likelihood that every symbol may occupy each position in the genetic sequence. However, the complexity of the LSP is prohibitively large, requiring many more solutions than could be stored to guarantee this condition.

The LSP is prohibitively complex and the number of symbols which could occupy a position in the sequence for the LSP has a factorial relationship with the number of population groups (K). This would require that a computer have sufficient memory to store a population

which grows in a factorial relationship to K and that the computer generates the solutions to occupy that memory – which in itself is problematic in terms of time.

Mutation and recombination allow for the chance that new symbols are generated after initial diversification, thus one may initiate the algorithm with a smaller population size whilst still allowing for the possibility of finding the optimal solution. It is important to note that no guarantee can be made for the optimality of the solution which is generated, but that without iterative diversification – and assuming that the population size is not at least $K!$ in size – one cannot guarantee that the optimal solution can be derived from the initial population.

DePermute also employs elitism and parallel cooperation as measures to improve on the iterative application refinement to solutions. Parallel cooperation is implemented by overwriting the least promising solutions of one solvers solution set and elitism ensures that a solution sets local best solution increases monotonically in optimality.

The following sections first provide an abstract description of DePermute and then introduce concrete details of the algorithm. This includes the introduction of a preprocessing step designed specifically for the LSP in the context of population structure studies which aims to reduce the complexity of the problem before beginning.

4.3.1 Algorithmic Model

Parallel solvers require the specification of:

- a metaheuristic search algorithm for performing local searches;
- the definition of a communications channel;
- a search space definition which supports partitioning;

Using the parallel solvers model, the l and i parameters are set to 0 and the S and n parameters are set to non-zero values in order to create a parallel genetic algorithm. This model leads to trivial parallelism in both the Refinement and Diversification operations. *Figure 4.1* illustrates a high level view of the resulting algorithm. Details regarding the algorithm follow.

Metaheuristic Specification

There are two mechanisms of Diversification in a genetic algorithm – both are embarrassingly parallel. The first is the creation of an initial population and the second is the mutation of members of the population.

Initial Diversification is, in essence, the generation of thousands of random numbers. Random numbers can be generated by many random number generating threads in a program and used together provided that the random number generators are not correlated.

Mutation is a simplified trajectory search mechanism which involves the same fundamental operation as the generation of the initial population. Mutation proceeds as follows in *Listing 4.7*. Also note that the specification breaks from the general model of trajectory Diversification for convenience.

For population-based Refinement, the elitism and crossover operators are selected. To implement these requires a four stage process. Given the count of elitist selections E and population size P we proceed as described by *Listing 4.8*.

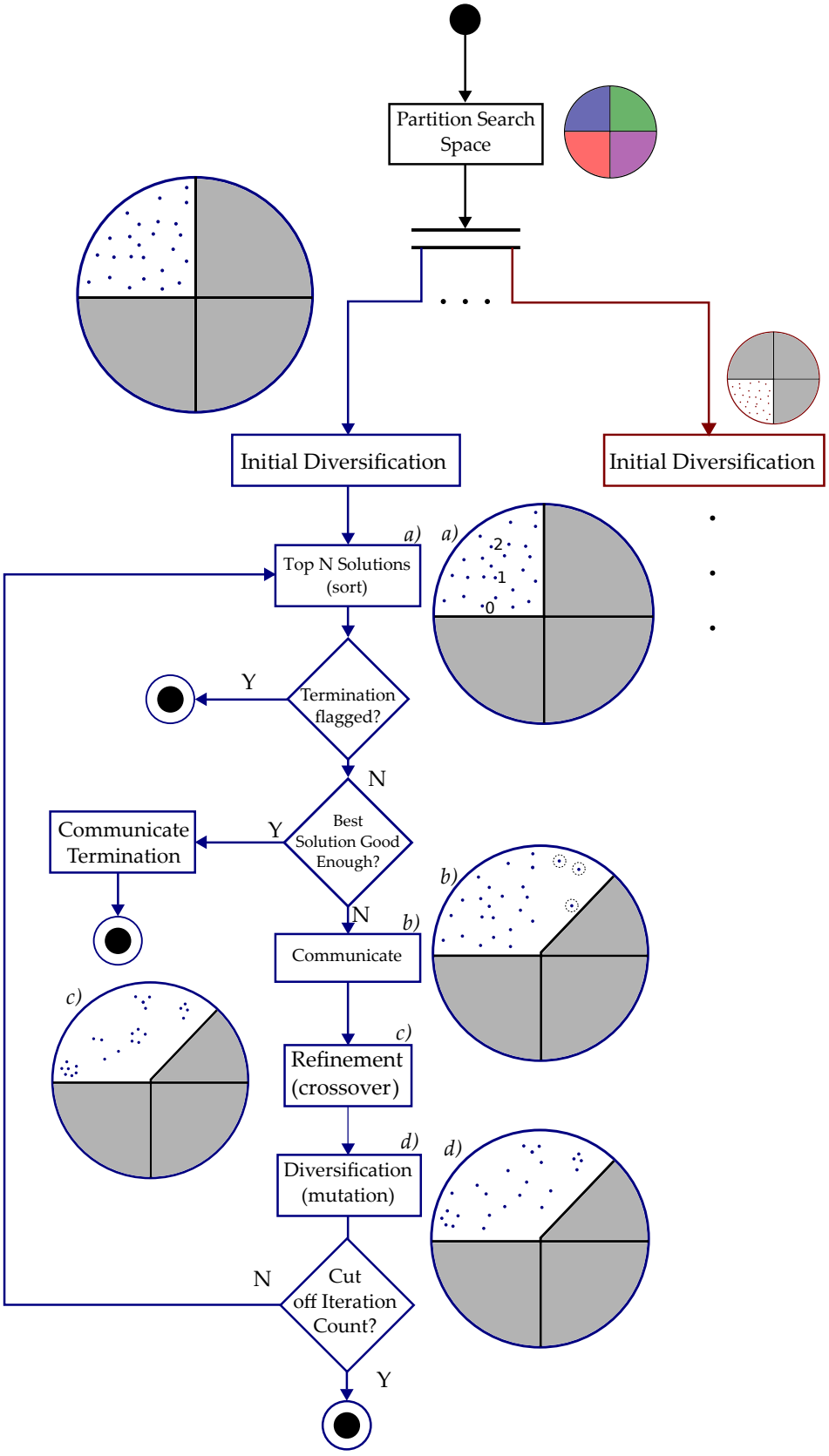


Figure 4.1: Illustration of the parallel genetic algorithm using the parallel solvers model.

Listing 4.7: Algorithm to perform mutation

```
// given:
//  – knuthShuffle () // A method which shuffles the given array
                      // resulting in an arbitrary permutation of
                      // the array

for solution in population (except elite):
    if (randomFloat(0, 1) < mutationRate)
        row <– randomInt(0, R – 1)
        knuthShuffle (solution[row])
```

Listing 4.8: Algorithm to perform crossover

```
fitnesses <– map F currentGeneration
sortByFitness ( currentGeneration, fitnesses )
cdf <– cdf ( fitnesses )

nextGeneration[0:E] <– currentGeneration[0:E]

for l <– E..P
    x          <– randomInRange [0, 1]
    indivOne   <– mapToCDF x currentGeneration
    indivTwo   <– currentGeneration[randomInRange[0, P)]
    crossPoint <– randomInRange [0, R – 1]
    nextGeneration[l][0:crossPoint] <– indivOne[0:crossPoint]
    nextGeneration[l][crossPoint:R] <– indivTwo[crossPoint:R]
```

Note that there is a distinction between the common mode of operation for Population Refinement and the selected mode. Implementations found in the literature specify that two solutions should be sampled under the CDF, however *Listing 4.8* reflects that only one solution is selected under the CDF while the other is selected completely at random.

Early experimentation suggested that the selection of two solutions under the CDF forced early convergence and as a result the search often became trapped in local minima. When one of the solutions is selected randomly, the convergence to the global minimum often occurred in markedly fewer iterations because the search was less prone to becoming stuck.

A buffer of the n best solutions is maintained without any modification to the algorithm because the sorting and elitism routines maintain a list of the best solutions at the head of the population. So long as n is less than E no extra storage or work is required in order to maintain the n best solutions.

Communications Channel

Two attributes are fundamental to the specification of a communications channel:

- how the topology of communications is structured, i.e. where connections exist between entities and what the direction of the connection is;
- how the flow of control is implemented, i.e. what is the protocol under which the solvers will communicate;

All-to-all communication has been described as desirable (e.g. [33]) but this may not be the case. Firstly, the communication of all solutions might cause solvers to converge on a small area of the search space too quickly, thus potentially conducting too sparse a search of their initial search space. Secondly, the overheads in such a communication scheme could be too great to warrant the benefits.

DePermute uses a simple neighbour based communication scheme. The communications channel therefore operates by making the population of each solver available to each other. An alternate description of the network is a ring bus topology where each solver sends a message to its neighbour at each instance of communication.

Search Space Definition

We define the $R \times K$ matrix as the search space. The empty matrix contains all solutions to the LSP. The matrix can have partial solutions entered into it to partition the search space, therefore the partition with an entire solution entered is the unit search space, i.e. a single solution. When entering values into the matrix the following two rules must be observed:

1. Only the symbols in $[0 \ldots K-1]$ can be entered.
2. A symbol already in a row cannot be reentered.

Solution Alignment

Jakobsson et al. define a solution as a collection of symbols in a matrix [24]. The rows of the matrix correspond to the result sets for model based population structure programs. The vertical arrangement of symbols in this format implies that the columns which the symbols

$$\begin{bmatrix} S_{0,0} & \cdot & \cdot & S_{0,K-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ S_{R-1,0} & \cdot & \cdot & S_{R-1,K-1} \end{bmatrix}$$

Figure 4.2: A general solution to the LSP in matrix format

represent are intended to have the same label. An illustration of a general solution matrix is depicted in *Figure 4.2*.

Each row of the solution matrix is an arrangement of symbols which align to symbols in other rows. Suppose that there exists a single solution, *optimal*, which is computed using the brute force algorithm for the LSP. Then, for the purposes of further analysis, a correct alignment is defined as follows.

Definition
Given a Q-matrix, Q. For each row Q_n ($n \in [0, .., R-1]$) symbol $Q_{n,i}$ is considered *correct* if it's paring with the symbol in row m, $Q_{m,i}$, is determined to be a pairing of symbols which exists in the *optimal* solution. Where $m \in [0, .., R-1]$, $i \in [0, .., K-1]$ and $m \neq n$.

This does not reflect the reality of what the Frobenius Norm heuristic would consider a "correct" alignment. Two columns can be considered correctly aligned (in the sense of the Frobenius Norm heuristic) if the distance between them is minimised, but there may exist another column in the same row as the second column's row which has the same distance from the first column. One may be able to distinguish which (between the competing alignments) is correct using biological evidence. However, in a mathematical sense the two are indistinguishable. It might also be the case that the correct pairing isn't that which has the lowest distance from a mathematical standpoint, because in certain circumstances a better alignment could be made if a poorer alignment is selected for one of the pairings.

Since the distinction from a mathematical standpoint is arbitrary, the alignment which is in the optimal solution (as found by the brute force algorithm) is defined as the *optimal* alignment.

Search Space Partitioning

There are two requirements for a search space partitioning scheme to be suitable for the algorithm:

1. The partitions which it creates must be distinct.
2. The un-partitioned search space must contain all possible solutions to the problem.

We define a partitioning algorithm for the search space as follows:

Definition

> Given a search space, in matrix format, locate any row of the search space which contains empty positions. If there are x empty slots in a row of the solution matrix then the search space can be partitioned x times by inserting each of the remaining symbols into the first empty slot (subsequently it may be partitioned $x - 1$ times by inserting each of the remaining symbols into the next slot of the same row.)

Note that each of the resulting partitions create solutions which are distinct from the solutions created by other partitions, because in all other partitions the symbol in the position which was set cannot be that same symbol.

Also note that continuing to partition a single row of the solution matrix by this process results in $K!$ partitions of the search space each with a complete first row. Continuing to partition each subsequent row results in $K!$ more partitions for each of the original partitions.

By the same argument, continued partitioning will ultimately result in $K!^{R-1}$ partitions, each containing a single solution to the problem. In addition, because each partition must be distinct, and because there are $K!^{R-1}$ solutions to the LSP, the partitioning scheme does not ignore any solutions to the LSP.

4.3.2 Concrete Algorithmic Description of DePermute

In *Section 4.3.1* DePermute is described on a high level in terms of the metaheuristic, communications channel and solution partitioning scheme it employs. The discussion now moves toward a more thorough treatment of the algorithm with the goal of elucidating the finer details of heuristics and discussing computational complexity.

Block Alignments Heuristic

The objective of the LSP in the context of population structure studies is to compute a consistent relabelling of columns in the result set of multiple runs of a model based population structure program. Upon closer examination of the Q-matrices which result from model based population structure programs it is apparent that there are trivial cases of the LSP.

Recall that the entries in the rows of Q-matrices correspond to probabilities and must therefore sum to one. If the value in index one of row one is x then the subsequent values

must be divided from the remaining $1 - x$. Furthermore, if x is very high, say for example 0.7, then the remaining values must be divided from 0.3. The highest value that a subsequent symbol can take on is therefore less than or equal to 0.3 in this instance.

In this instance it is likely that the individual whose matrix contains this value is not very admixed with the other population groups in the study. It may be deduced that this individual has a similar value in every other matrix in some column. I have named the arrangement of symbols by this method a "block alignment." *Figure 4.3* illustrates the process of finding a block alignment in a collection of matrices.

One can be fairly certain that if x is sufficiently high and, for the same individual, there exists a value close to x in all other runs then the columns in which those values occur ought to have the same label. Block alignments can fill in a column of the solution matrix for each instance which is found.

The rules for selecting a block alignment are as follows. If a value x exists in an Q-matrix A which is greater than σ then for each other Q-matrix there must exist a value y in the same row as x which is within $\pm\delta$ of x in order for the columns in which x and the y values exist to be block aligned. *Listing 4.9* delineates the algorithm for selecting block alignments in R, $m \times K$ Q-matrices.

Based on the bounds of the loops it would appear that the worst case running time for this algorithm is $O(MRK^2)$. However, if the physical constraints of the data are taken into account then it becomes apparent that this cannot be the case. This is true because a value of σ greater than 50% renders it impossible to find more than one possible alignment in a row of the first Q-matrix, because all other values must be divided between the remaining $< 50\%$. Taking this into account the worst case running time becomes $O(M(K + RK))$ because one can find and check at most one potential hit per row of the first Q-matrix.

For each block alignment a column is added to and fixed in the solution. Therefore, the width of the solution matrix K is (in effect) reduced by one for each alignment. If x block alignments are made then the complexity of the problem to be solved by parallel solvers becomes $O(K - x)!^{R-1}$, thus significantly reducing the complexity of the problem.

Branch and Bound

The partitioning process is implemented with a Branch and Bound algorithm. The intention of the algorithm is to avoid partitions of the search space which contain poor quality solutions prior to starting the iterative stage of the algorithm. An outline for the Branch and Bound algorithm is as follows:

- Construct a queue of partitions; initially with the complete search space as the sole element.
- while the queue length is less than the desired number of partitions:
  - Pop from the queue.
  - Partition the popped search space as many ways as the first non-fixed symbol allows.
  - Enqueue each sub-partition.
  - Estimate the upper and lower bound fitness for each partition in the queue.

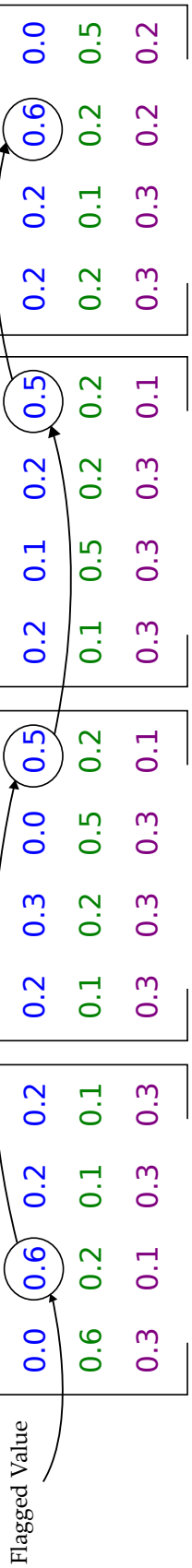


Figure 4.3: Illustration of the process of finding block alignments

Listing 4.9: Algorithm for selecting block alignments

```
// Given: matrices (the set of Q matrices)

solution <- []
// For each index in the first Q-matrix
for j <- 0..m - 1
    for i <- 0..K - 1
        // High value found?
        if matrices[0][j][i] > sigma
            alignment <- []
            // For each index in each row of every other Q-matrix
            for k <- 1..R - 1
                aligned   <- false
                attempted <- false
                for l <- 0..K - 1
                    // Value in other matrix which works?
                    attempted <- true
                    if matrices[k][j][l] > (matrices[0][j][i] - delta) and
                       matrices[k][j][l] < (matrices[0][j][d] + delta)
                        alignment <- alignment + [l] // i.e. append the inner
    loop variable, 'l'
                        aligned <- true
                        break
                // One of the other matrices didn't have a similar value?
                if not aligned and attempted
                    alignment <- []
                    break
            // Check whether alignment can be added to the solution
            if alignment != []
                canAlign <- true
                for k <- 0..K - 1
                    for l <- 0..R - 1
                        if solution[l][k] == alignment[l]
                            canAlign <- false
                if canAlign
                    solution <- solution + alignment
```

– If there exists any partition whose lower bound estimate is higher then every other upper bound, then remove that partition from the set.

It is difficult to provide accurate bounds on the complexity of this algorithm. Instead it will be parameterised abstractly, in terms of the cost of finding the next element to partition, making K partitions and estimating bounds. Estimating bounds on a partial solution is depicted in *Figure 4.4* and is achieved as follows:

1. The cost of columns which have been entirely filled in is evaluated.
2. The cost of partially filled columns is evaluated twice in order to establish an upper and a lower bound of the cost of the partition:
   (a) by iteratively entering the remaining elements which would result in the lowest increase in cost (i.e.the "best greedy selections");
   (b) by iteratively entering the remaining elements which would result in the highest increase in cost (i.e.the "worst greedy selections");

Suppose that the complexity of finding the next element to partition is $O(G(n))$ and that the complexity of evaluating a columns fitness is $O(F(n))$. Finally, at most $\lceil \log_k n \rceil$ partitions must be made to make at least n partitions of the search space. Therefore, the complexity of the Branch and Bound algorithm is $O(G(n)nF(n) \log_k n)$.

Parallel Solvers Detail

The parallel solvers algorithm presented in *Section 4.3.1* is abstract because it doesn't elucidate details regarding the algorithm, for instance how fitness is computed, or how fitnesses and solutions are sorted. In this section a more thorough treatment of the algorithm is presented in terms of operations such as fitness evaluation. The following algorithms will be presented:

A. Fitness Evaluation;

B. Sorting;

C. Cumulative Distribution Function Creation;

D. Crossover;

E. Mutation;

F. Recombination (an experimental extension);

A. Fitness Evaluation
We define two fitness functions:

1. the average pairwise euclidean distance between all Q-matrix pairs;
2. the average pairwise euclidean distance between the first Q-matrix and all subsequent matrices;

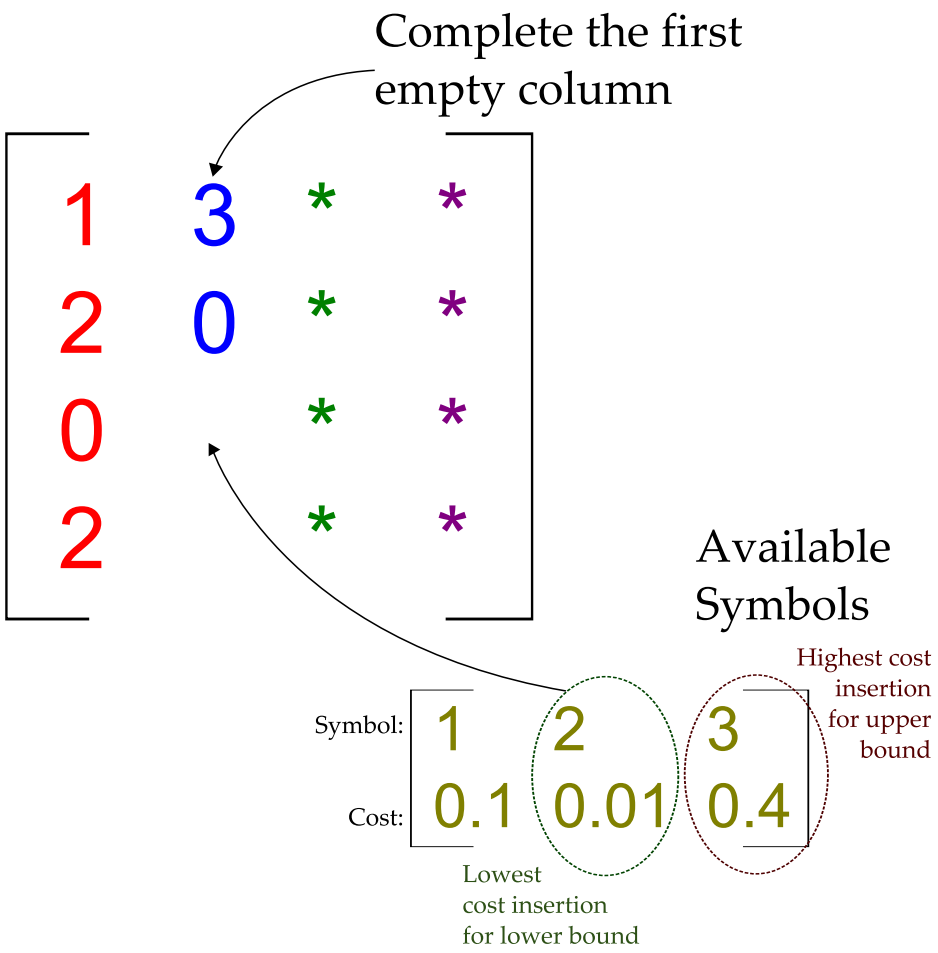


Figure 4.4: Illustration of the greedy cost algorithm (the depicted insertion is one of two needed to compute the lower bound)

Listing 4.10: Algorithm to build the cache of column distances

```
for l <- 0..R - 2
    for L <- l + 1..R - 1
        for ll <- 0..W - 1
            for LL <- 0..W - 1
                distance <- 0
                for lll <- 0..MD
                    distance <- distance + ( matrices[l][ll][lll] -
                                             matrices[L][LL][lll] )^2
                cache[l][L][ll][LL] <- distance
```

Listing 4.11: Algorithm to compute the fitness of a solution

```
for i <- 0..C
    fitness <- 0
    for j <- 0..W - 1
        for k <- 0..D - 1
            for l <- k + 1..DEPTH - 1
                fitness <- fitness + retrieveFromCache ( i, j, k, l )
    fitnesses[i] <- fitness
```

The second fitness function is an approximation of the first and can be used in instances where the problem size is very large to find a reasonable approximation of the answer. The equations for these functions are given as follows.

$$F_{acc} = \frac{\sum_{i=0}^{R-2} \sum_{j=i+1}^{R-1} F_{norm}(M_i, M_j)}{0.5R(R-1)} \tag{4.1}$$

$$F_{app} = \frac{\sum_{i=1}^{R-1} F_{norm}(M_0, M_i)}{R-1} \tag{4.2}$$

To compute the average pairwise Frobenius norm requires that the sum of the euclidean distances between columns in each pairing of Q-Matrices be computed for a given arrangement – as described by a solution.

We note that computing the distance between columns every time a fitness is evaluated is cumbersome. Therefore, a cache is built up with the distance between all possible pairings of all columns in different matrices. Cache construction is achieved by the algorithm given in *Listing 4.10*. Note that the square root is omitted in the algorithm because the square root in the Frobenius norm is taken after the sum of the squares of differences between column values.

We define the algorithm for computing the fitness of a solution using four nested loops. This algorithm corresponds to the first fitness function, the second can be implemented by setting `l` to 0 and removing the outermost loop.

B. Fitness Sorting

Sorting is achieved by a comparison based sorting algorithm with running time propor-

Listing 4.12: Algorithm to compute the CDF

```
acc <- 0
for l <- 0..P
    acc    <- acc + fitnesses[l]
    cdf[l] <- acc

for l <- 0..P
    cdf[l] <- cdf[l] / acc
```

tional to $O(n \log_2 n)$. In practice a custom version of the algorithm is required because the standard C++ implementation cannot be used to sort key-value pairs in two separate buffers. The "bottom up" merge sort algorithm was chosen because it appeared to be inherently parallel; however the use of a faster algorithm, such as radix sort or quick sort, would have been better in practice because the CPU implementation does not leverage parallelism in sorting. It was decided that the use of the simpler "bottom up" merge sort algorithm was acceptable because the complexity of sorting is much smaller than that of other steps in the algorithm.

C. Communications

Once the solutions have been sorted, they are communicated. Communication is a simple copy operation with running time proportional to $O(nRK)$ where the parameters n, R and K are a result of the number of solutions communicated and the bounds of the solutions matrix.

D. Cumulative Distribution Function Computation

Computation of a Cumulative Distribution Function (CDF) is achieved by a two-pass algorithm which first runs a scanning sum across the fitness values and then normalises the resulting buffer with the last value of the scanning sum. This algorithm is illustrated in *Listing 4.12*

E. Crossover

The algorithm for crossover is as follows. First E solutions are copied into the next generation. Then for the remaining indices a random number in $[0, 1]$ is generated and a binary search is conducted on the CDF in order to find the solution whose fitness contributed to the CDF reaching that value. A second solution is picked at a random index in $[0, P-1]$ where P is the population size.

A random integer x is selected in $[0, R-1]$ and used as the crossing point between solutions. To cross the solutions, the rows up to x are copied from the first solution into a new solution in the next generation and the rows including and subsequent to x are copied from the second solution into a new solution in the next generation. *Figure 4.5* illustrates the process of selecting two solutions and then crossing them to create a "child" solution.

Listing 4.13 illustrates the algorithm for performing crossover on the population. This algorithm has time complexity of $O(EKR + (P-E)(\log_2 P + KR)) = O(PKR + (P-E)\log_2 P)$.

F. Mutation

Solution mutation is presented in *Section 4.3.1* this algorithm has worst case complexity of $O((P-E)R)$, i.e. when every solution in the population is mutated. $(P-E)$ results from performing mutation on every non-elite solution and the R factor results from the complexity of the Knuth Shuffle [28].

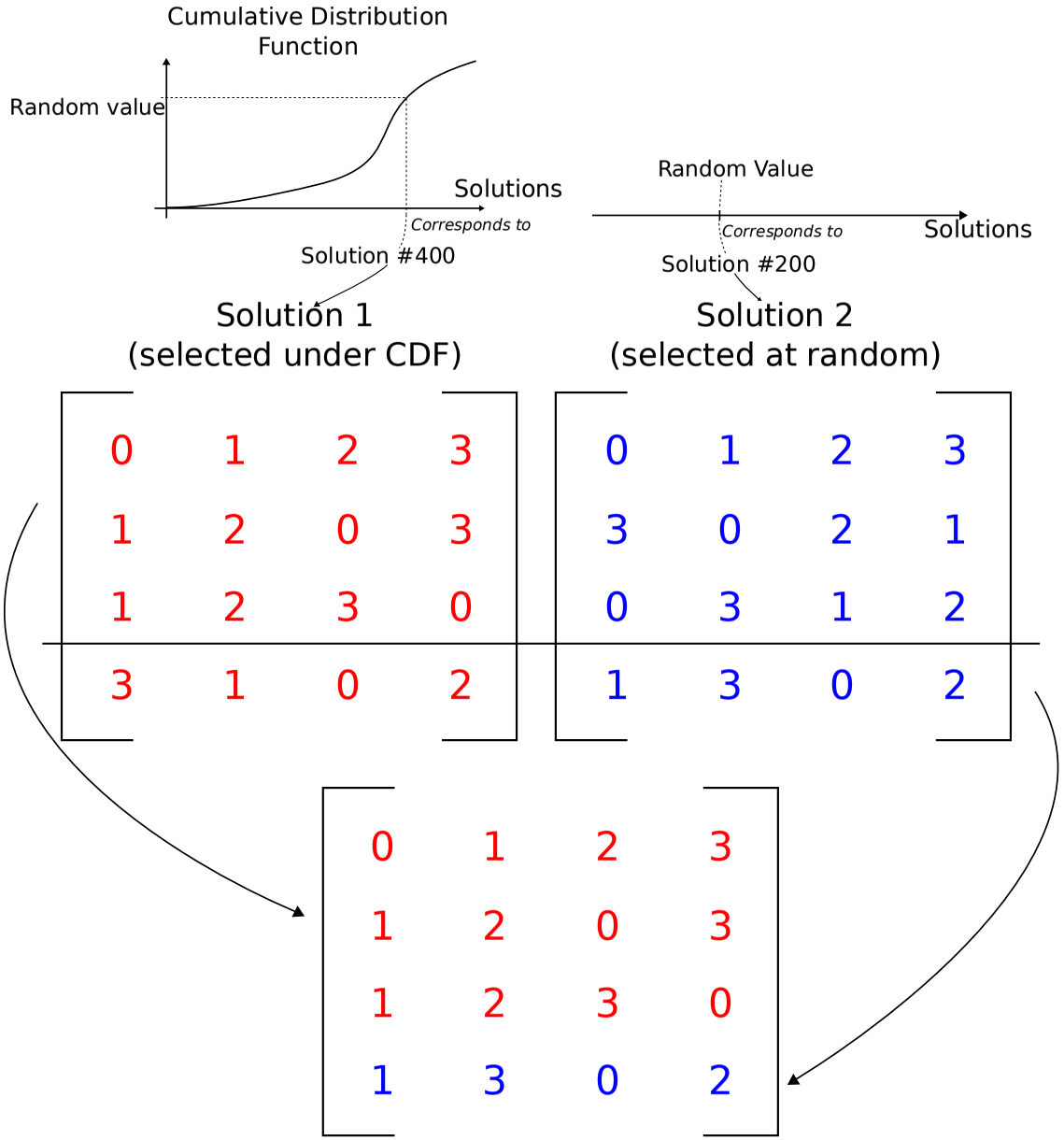


Figure 4.5: Illustration of crossing two solutions with single point crossover. The first solution is selected randomly with a bias towards solutions with a high fitness and the second is selected at random.

Listing 4.13: Algorithm to carry out crossover

```
for l <- 0..E - 1
    nextGeneration[l] <- copySolution ( population[l] )
for l <- E..P - 1
    solution1     <- binarySearch ( random [0, 1], cdf, population )
    solution2     <- population[random[0, P]]
    crossingPoint <- random [0, D)
    for l <- 0..crossingPoint
        for L <- 0..K
            nextGeneration[l][L] <- solution1[l][L]
    for l <- crossingPoint..D
        for L <- 0..K
            nextGeneration[l][L] <- solution1[l][L]
```

Listing 4.14: Algorithm to recombine a portion of the population

```
for l <- E..E + C - 1 // The first `C' individuals in the population after the
    elite
    crossingPoint <- randomInt ( 0, R - 1 )
    for L <- 0..K
        for ll <- 0..crossingPoint
            nextGeneration[l][L][ll] <- population[l][L][K - crossingPoint + ll]
        for ll <- 0..R - crossingPoint
            nextGeneration[l][crossingPoint + ll] <- population[l][L][ll]
```

G. Recombination

Recombination is an experimental extension to the algorithm. It is inspired by the process of recombination in nature. Recombination in DePermute is a simplified model of the mechanism found in nature and in the context of combinatorial optimisation it can be described as continued population-based diversification.

The algorithm proceeds as in *Listing 4.14*. For each individual after the elitist section of the population and up to the recombine count; swap a portion of the solution from the top with its bottom portion, where the swap position is a random number in $[0, R-1]$. This procedure has complexity $O(CRK)$ (C for the count of recombined individuals, RK for the complexity of copying a solution).

There are two desired effects of this operation on the population of solutions. The first desired effect is a more diverse population. This results in the expansion of the search and potentially in the escape of local optima. Secondly the per row aggregate content – the set of orderings which exist on a per row basis across all solutions – of the set of solutions is changed.

If there are $K!$ orderings of rows then for it to be possible that every permutation of the symbols exist on the same row across all solutions then at least $K!$ solutions are required. This number is prohibitively large and doesn't guarantee that all orderings are randomly generated. If an ordering in the optimal solution for any given row is not generated in initial

diversification on the same row in some solution, and there is no iterative diversification, then it becomes impossible to find the optimal solution.

In practice one relies on mutation to, per chance, mutate a row and have that mutation be selected for. However, the desired ordering might exist in another solution in the population already. If a portion of the population is recombined then we allow for the chance of the correct symbol in the wrong row to move into the correct row. In a subsequent section, this intuition is tested to determine the effect of the operator.

Overview of the Steps Involved in the Genetic Algorithm

It's worth noting that DePermute bears close resemblance to a standard genetic algorithm. The similarities are as a direct consequence of a bias towards the degree of population diversification which has been selected for. Notable dissimilarities lie in the algorithms use of a custom model for the search space (discussed further in a subsequent section) and in the way it communicates without blocking. DePermute also implements recombination and could have modified the behaviour of mutation.

Recombination doesn't exist in a traditional genetic algorithm and represents a further opportunity for population diversification to occur. In a traditional genetic algorithm the only operator on the population scale is crossover. Crossover is an inherently destructive operator as it reduces the diversity of the population on which it operates. Recombination ought to provide the opposite effect by presenting the algorithm with more candidates to select from.

DePermute uses a simplified version of trajectory diversification. The Parallel Solvers model allows for trajectory diversification and had it been implemented in DePermute then it would have modified the way that Mutation occurs. i.e.mutation could have been implemented (with diversification in mind) by constructing n mutated copies of a given solution. Trajectory refinement then selects a solution from the generated candidates to replace the original solution.

4.3.3 Data Structures

The discussion now focuses on the data structures which are processed by DePermute with particular focus on their memory footprint and on their limitations. There are four data structures used by DePermute, they are:

- the solution structure;
- a special space and time efficient division structure termed the "Leaf Root Tree";
- the sub-space search structure;
- the column distance cache (which is discussed further in *Chapter 6*);

Solution Structure

Three representations of a solution data structure have been devised. These are presented in increasing levels of space efficiency and decreasing levels of time efficiency.

Structure One

The first representation of a solution is the matrix representation defined in *Section 4.3.1.* The matrix representation requires memory proportional to $O(KR)$ and has access time proportional to $O(1)$. The matrix representation is convenient for fast fitness function evaluation and is human readable.

Listing 4.15: Algorithm to compute the *nth* permutation of a collection of symbols

```
// Given: n   (an integer corresponding to the position of a permutation
//             in an ordered arrangement of all permutations)
//        g() (A function which generates the `next' permutation of a sequence
             of symbols)

nthPermutation <- symbols
for l <- 0..n - 1
    nthPermutation <- g nthPermutation
```

Structure Two

The second representation of a solution is a solution as an ordered collection of integers. Each integer in the collection corresponds to the position of a permutation of symbols in an ordered arrangement of all permutations, e.g. the number zero corresponds to the 0^{th} permutation of the symbols in a row of Structure One. This representation is space efficient, requiring only $O(R)$ storage. However, in order to access the symbols in a row, one must generate the permutation corresponding to that row's integer.

This can be achieved by enumerating permutations and then selecting the n^{th} permutation. This algorithm executes in time proportional to $O(K!)$ and requires an additional $O(K)$ to temporarily store the corresponding permutation. According to Stirling's approximation $K! \approx (\frac{k}{e})^k \sqrt{2\pi k} = O(k^{k-\frac{1}{2}} e^{-k})$, thus the algorithm completes in time proportional to $O(k^{k-\frac{1}{2}} e^{-k})$ [47]. The algorithm for determining the n^{th} lexicographical permutation of a given set of symbols is given in *Listing 4.15*.

Structure Three

The third representation of a solution is a solution as an integer. The integer represents the position of the solution in an ordered arrangement of all solutions. This representation is the most space efficient, requiring only $O(1)$ storage, but requires the most work to extract the ordering of columns.

An ordering for solutions can be established by arbitrarily defining solution zero as the solution containing ordering zero for all rows. The next solution has ordering one in the first row of the solution matrix and ordering zero in subsequent rows. One can define incrementation for solutions as proceeding to the next ordering for the zeroth row and if that row becomes the zero permutation again, recursively incrementing the subsequent row by the same method and so forth.

The solution can be thought of as a number with factorial base because $K!$ increments are made to the first row of a solution matrix to overflow the next row.

A linear time algorithm ($O(R)$) can be used to convert the integer representation into a collection of orderings. The algorithm proceeds as illustrated in *Listing 4.16*.

After the transformation in *Listing 4.16* the solution will be in the format of Structure Two and can be converted to the matrix representation in $O(RK)$ time.

Analysis of Efficiency

Having proposed three data structures which could be used to represent a solution, the following sections examine those data structures in terms of the trade offs they make between time and memory. Importantly, the choice is contextualised in terms of the algorithmic steps

Listing 4.16: Algorithm to transform representation 3 to representation 2

```
Given: R & k
       n     // A number to convert

orderings <- []
divider   <- K!^(R - 1)

for l <- 0..R
    orderings <- [n mod divider] + orderings
    n         <- n div divider
    divider   <- divider / K!
```

involved in DePermute.

Mutation

Firstly the choice is evaluated against the mutation algorithm.

The mutation of solutions in the second format is trivial and can be achieved by replacing an integer at a random position with a random integer in $[0, K!)$. In the case of the third representation, the integer must first be converted into the second representation, mutated according to the mutation scheme for the second representation, and then converted back into an integer. In the case of the third representation, a row must be selected at random and then the Knuth Shuffle[2] used to mutate that row to a random row.

The second representation is the most efficient saving a factor of $O(K)$ in time and $O(K)$ in space over the first representation. The third representation saves a factor of $O(RK)$ in space over the first representation, R over the second and runs in the same time as the first representation – $O(K)$.

Fitness Evaluation

Secondly the choice is evaluated against the fitness evaluation algorithm.

The evaluation of the fitness of a solution requires that that the solution be in the format of Structure One. Structure Two requires $O(K!)$ for conversion to Structure One and Structure Three requires $O(K!K)$ for the conversion to Structure Two and then to Structure One.

The only advantage which the structures Two and Three present over Structure One is that only one solution needs to be converted at a time. Therefore, the Structure Two requires only $O(PK + RK)$ storage and Structure Three requires only $O(P + RK)$ storage, whereas Structure One requires $O(PRK)$ storage.

Crossover

Finally the choice is evaluated against the crossover algorithm.

Crossover for Structure Two is trivial and may be achieved by copying an integer per row in each corresponding solution into the new solution. This operation saves $O(K)$ over Structure One because there's no need to copy the elements in each row of the matrix as is the case with Structure One. The third representation requires an $O(K)$ transform to the second representation and then back again after carrying out the crossover operation.

Summary

Structures Two and Three save memory proportional to $O(K)$ and $O(KR)$. Structure Two

[2]The Knuth Shuffle is a simple and fast algorithm to produce a fair, random permutation of a list

is the most time efficient for mutation and crossover saving a factor proportional to $O(K)$, however it results in a cost of $O(K!)$ for fitness evaluation. Due to the extra overhead of computing Structure One from Structure Two, only Structure One is suitable for DePermute.

Leaf Root Tree

The partitioning scheme outlined in *Section 4.3.1* is not time efficient. It is desirable to achieve both time and space efficiency while maintaining the semantics of the partitioning scheme.

This dissertation presents the Leaf Root Tree (LRT) data structure for this purpose. The data structure is an efficient representation of what parts of a solution are fixed.

Recall that two symbols were said to be aligned if they appear in the same column in the solution matrix representation of a solution. Instead of inserting elements into empty entries in the rows in a matrix to generate new alignments, the LRT interprets the symbols as nodes in a graph and connects every node in one row to all nodes on the next row.

One column, as it is formulated in the matrix representation, can be constructed by taking a path from a node which hasn't yet been visited on the top of the graph, to the bottom of the graph, marking nodes as visited as they are passed and never passing through a node which has already been visited. In order to support the creation of entire solutions, a special node is added at the bottom and at the top of the graph which has the following properties:

- Special nodes are never marked as visited.
- The special node at the top connects to all nodes on the top row of the graph.
- The node on the bottom connects to the node on the top.
- All nodes on the last row connect to the node on the bottom.
- When a special node is visited it is not appended to the solution matrix.

A full solution can be constructed from an LRT by continuing to visit nodes until no available paths remain. A representation of the LRT which only indicates which paths remain is used in practice because the information about how the tree was traversed is intrinsic to the solution matrix constructed by the LRT.

The data structure which stores only the remaining paths can be represented as a matrix of binary values, in which nodes which have been visited are zeros and those which have not are ones. This leads to a space efficient representation of the paths remaining in an LRT as a collection of words. For example, if $K = 3$ and $R = 5$ then the paths in an un-searched LRT can be represented by a buffer of length 5 with all entries set to $2^3 - 1$. As nodes in the LRT are visited they're bits are unset. As an LRT is searched more of the bits will be unset until its representation is a buffer of zeros. An LRT representing an entire traversal of the tree, yielding a complete solution, along with its corresponding binary representation is illustrated in *Figure 4.6* for $K = 4$, $R = 3$.

An LRT is space and time efficient in the computation of random solutions within a partition of the search space. Recall from previous sections that generating a random solution in a solution matrix would involve searching for remaining options. This process is $O(KR)$; however one can identify the rows of a binary encoded LRT which contain un-visited nodes

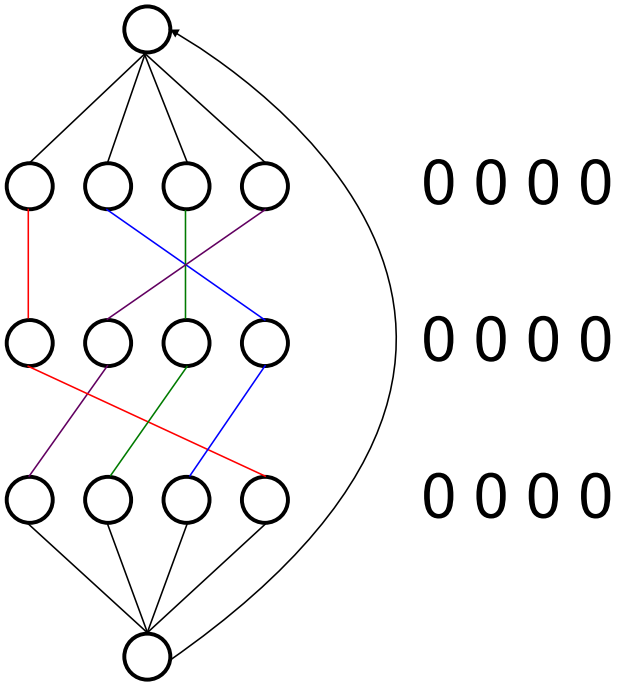


Figure 4.6: Illustration of a $K = 4$, $R = 3$ LRT and its corresponding binary representation. Note that because all the nodes have been visited its binary representation has all bits unset.

by checking that the integer on that row is greater than zero. Therefore searching for a row with an option to take is $O(R)$.

To take a path at random, one can generate a random number i in $[0, K-1]$ and select the index of the i^{th} set bit in the LRT as the next node in a path (marking that bit off). Continuing to do so will result in a complete solution. Therefore the LRT representation allows for solutions to be generated in matrix format in time proportional to $O(KR)$.

Sub Search Space

The binary encoding of an LRT is an efficient method of indicating what selections are available; however a partition of the search space requires that partial paths are taken. The partial LRT accomplishes this purpose. A partial LRT behaves in the same way as a conventional LRT but it is used to encode a partial path in the search space.

The partial LRT is explained by way of example. For $K = 4$ and $R = 3$, suppose that a search space is partitioned twice. These partitions are illustrated by graph LRTs in *Figure 4.7*.

The LRT illustrates the nodes which can still be taken but it cannot express a partial path. An alternative representation of the searching process is to indicate the path which has been taken thus far and the nodes which are still available. If this representation of the search is used in conjunction with a partial solution in matrix form then we may express partial paths as well as which paths are still available. This LRT is called a "partial LRT." *Figure 4.8* illustrates the partial LRTs which correspond to the second partition of the an LRT (as illustrated in *Figure 4.7*).

Finally, if any columns are completed during the partitioning process then the column is

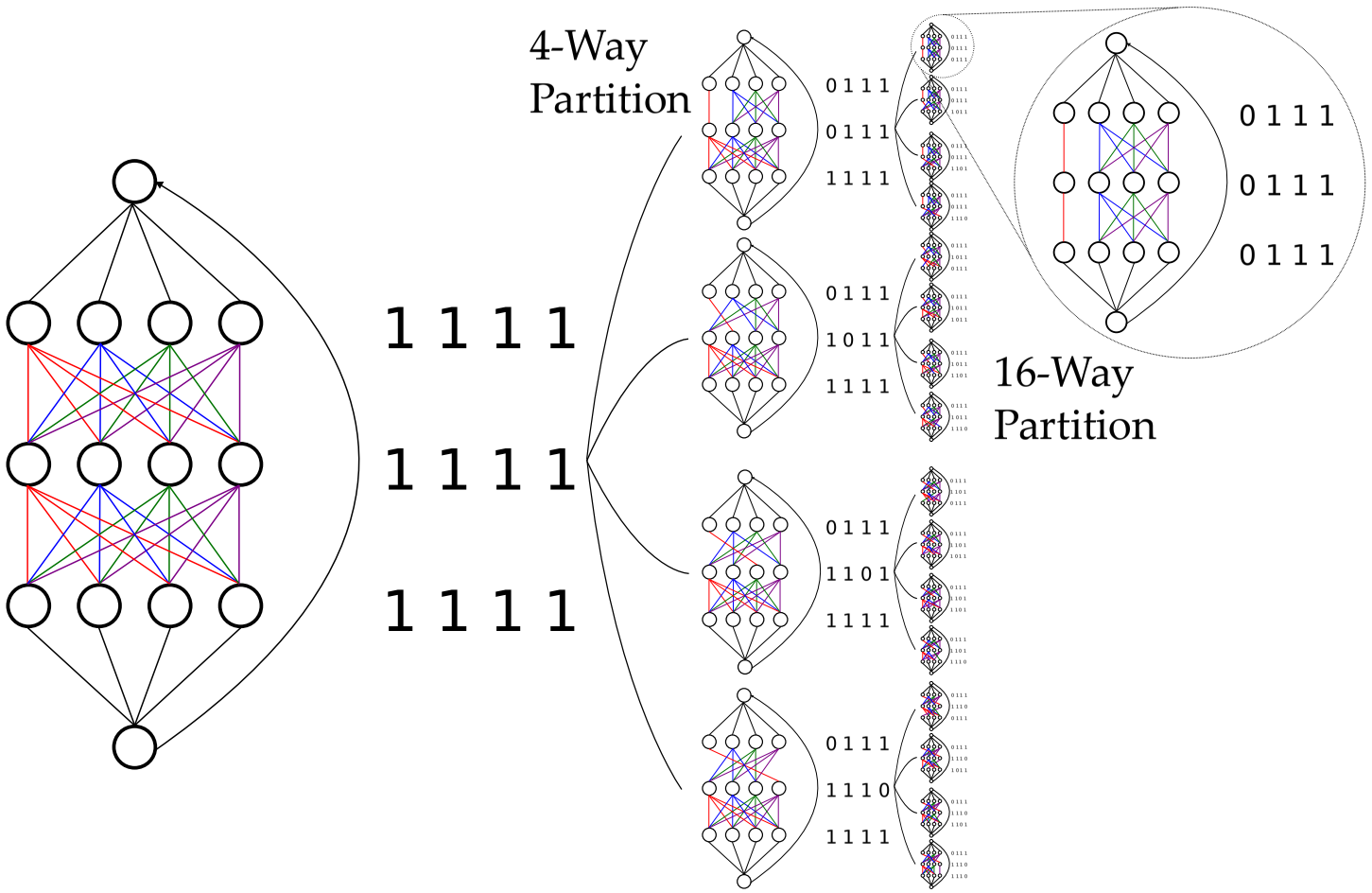


Figure 4.7: On the left an un-searched LRT is depicted (note that all paths are available and all bits are set.) As partitions are made bits are unset and one edge in the path is fixed – eliminating 3 alternative paths. Two partitions of the LRT at this stage yield 16 LRTs.

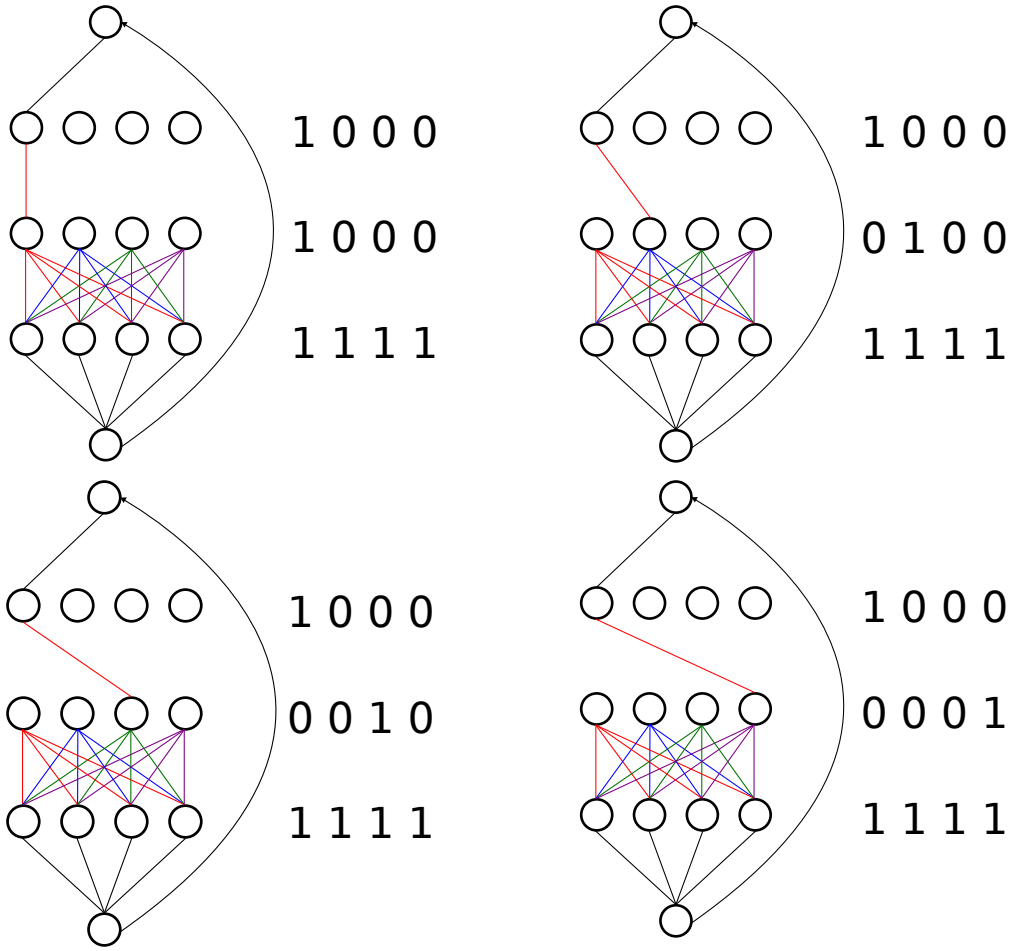


Figure 4.8: The partial LRTs corresponding to the LRTs in the first partition of *Figure 4.7*. Note that instead of un-setting only one bit in the first row, we now set only the bits corresponding with the terminals of the edge which was traversed.

added to a partial solution. The pairing of a partial solution with a partial LRT is considered a sub-search space. A sub-search space in which two columns have been completed is illustrated in *Figure 4.9*.

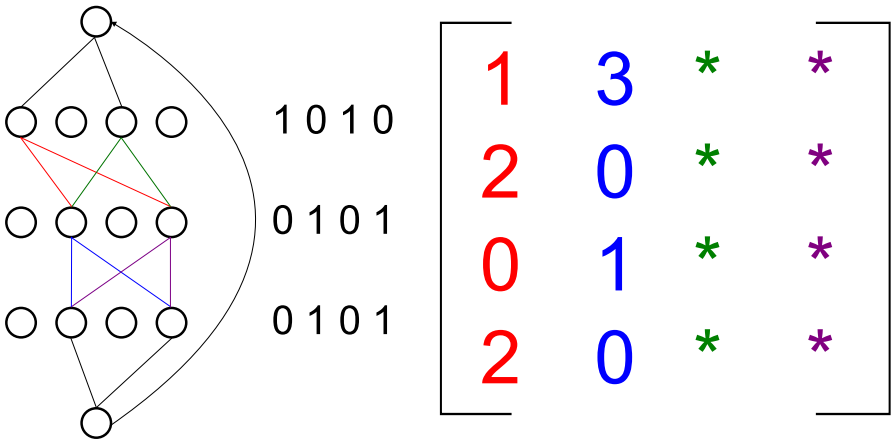


Figure 4.9: A sub search space in which two columns have been completed. The partial solution indicates the paths which have been taken while the partial LRT indicates those which may be taken next.

A sub-partition is faster to partition than a solution matrix on its own because one can search for a row with options remaining in it by checking whether the integer on that row of the partial LRT is a power of two. If it is a power of two then the selection is fixed and there aren't any options on that row. The complexity of performing this search is $O(R)$ whereas for the solution matrix it is $O(KR)$ because it must inspect every symbol entry to determine whether it is empty.

Chapter 5

Testing the Convergence Rates of Heuristics

5.1 Motivation

A heuristic is often employed to shorten the path which a program takes on its route to the solution. In this way heuristics can be thought of as rules of thumb which a program uses to make a problem more tractable. Heuristics can be implemented in a mathematical sense as equations which simplify the problem or in the logic of an algorithm by making assumptions about which decision is best.

This section presents experiments to determine how effective the heuristics employed by DePermute are in reducing the number of iterations which the algorithm makes before it finds a solution of suitable quality.

In reality the number of iterations taken to reach a suitable solution cannot be the only qualifier of a good heuristic, because a very complex heuristic may indeed reduce the number of iterations taken only to greatly increase the amount of time which is required to compute each iteration. The picture is completed in *Chapter 6* which uses time as the dependent variable.

It's important to note that the approach to testing adopted in this chapter does not respect the structure of real data sets. Instead it attempts to establish characteristics of the heuristics employed when supplied with entirely random data. This approach does not address the effectiveness of heuristics when supplied with real data. It was decided that this line of thought was outside the scope of this dissertation.

In this chapter convergence rate is measured in terms of the number of iterations DePermute makes before discovering a solution of sufficient quality. This testing methodology emphasises the efficacy of heuristics in the absence of running time. This is both important and relevant because program running time is dependent on many (often uncontrollable) variables.

Experiments designed to measure running time are sometimes unrepeatable, because insufficient detail of the experimental procedure has been documented. In addition; extraneous

effects such as the hardware which can be afforded, the competency of the researcher in software development and the time constraints on the project might lead to false conclusions. Therefore a robust, simple, simulation of DePermute is used in this chapter. In the absence of a parallel computer one can investigate the number of iterations which are required on average to compute satisfactory solutions to problems of varying complexities.

An investigation of this kind does not reflect the running time of an implementation of the algorithm, but it could help develop a more accurate model of the running time.

5.2 Test Design – Abstract Simulation

This section seeks to establish the effect of heuristics on convergence rates. This is achieved by simulating random input data sets to the LSP and tuning the DePermute program so that only one particular heuristic is active at a time. Several repeats of each experiment are run and the average number of iterations is measured in each instance.

5.2.1 Generation of Random Data Sets

An important aspect of the testing process is how problems are randomly generated. In practice, there are many effects which may occur in data sets extracted from test subjects. It may be difficult to simulate some of these effects or impractical in terms how long it would take to truly simulate certain effects.

The most important aspect of data in actual experimentation is that there would almost certainly be underlying structure, signifying the genetic heritage of an individual in the study. Structure in the data is significant from an algorithmic perspective because it affects the running time of a data driven algorithm, i.e. an algorithm which is able to identify structure and adapt a search so that it may terminate earlier would take advantage of structure, whereas algorithms which do not employ this principle would not be impacted by underlying structure.

The testing process will not simulate structure in the data in the interest of discovering the bounds of the computational characteristics of the algorithm. This methodology will establish a practical upper bound on the convergence rate. The lower bound, being as it is data driven, is near one iteration of the algorithm.

The process which is employed in this chapter – and throughout this dissertation – for random data set generation is as follows:

1. Generate a random Q-matrix of an appropriate size, and then:

   Repeat the following M times (fixed to 100 for the following experiments experiment):

   (a) Sample K random numbers in $(0, 1)$ under a uniform distribution (i.i.d.).
   (b) Normalise the random numbers.
   (c) Append the random numbers to the prototype matrix.

2. Copy the prototype matrix R times.

3. For each value in the matrix add a random real number x, where $x \in (0, n)$.

4. Normalise the rows of each matrix again.

5. Perform a Knuth Shuffle on the columns of each matrix.

5.2.2 Population Diversification

In terms of the DePermute algorithm exploration is dependant on the initial Diversification process. Initial Diversification in a population-based metaheuristic creates a so-called "primordial pool" of genetic material from which the optimal solution can be found by some combination of the initial components.

It is pertinent to ask how fast the algorithm converges as a function of the initial population size. In order to isolate this effect all other sources of Diversification are disabled because these effects serve the same purpose as initial Diversification – which is to explore a greater amount of the search space. In an effort to keep the experiment simple, only crossover is used in terms of Refinement of solutions.

At small population sizes there may not be sufficient aggregate row content across solutions for the correct solution to be formed as a combination of initial solutions. Therefore, one would expect the number of iterations to tend towards infinity as the population size decreases. Likewise it is expected that there be a single iteration of the program when population size exceeds the number of possible solutions.

Expected Mathematical Relationship
The convergence rate has an hyperbolic relationship to the initial population size.

Experiment 1 (see below) is designed to test the response of the algorithm to varying population sizes.

Experiment 1
Population Diversification Experiment:

1. Generate n random problems for increasing problem sizes (K, R).
2. Disable sub space division, block alignments, elitism and mutation.
3. Run DePermute three times for each problem, and for each population size interval value.
4. Average the number of iterations taken by the algorithm to reach at least 0.01 fitness (a near perfect solution).

Results

The results confirm our expectation that the rate of convergence on an high quality solution has an hyperbolic relationship to the initial population size used in the algorithm. In all instances of the experiment any incrementation of the population size becomes ineffectual after a certain point. Increments after this point yield insignificant decreases in the number of iterations to achieve an high quality solution.

5.2.3 Mutation (Trajectory Diversification)

Mutation is designed to expand the scope of the search the algorithm during iterations. This is achieved by randomly perturbing solutions so that the aggregate row content of solutions

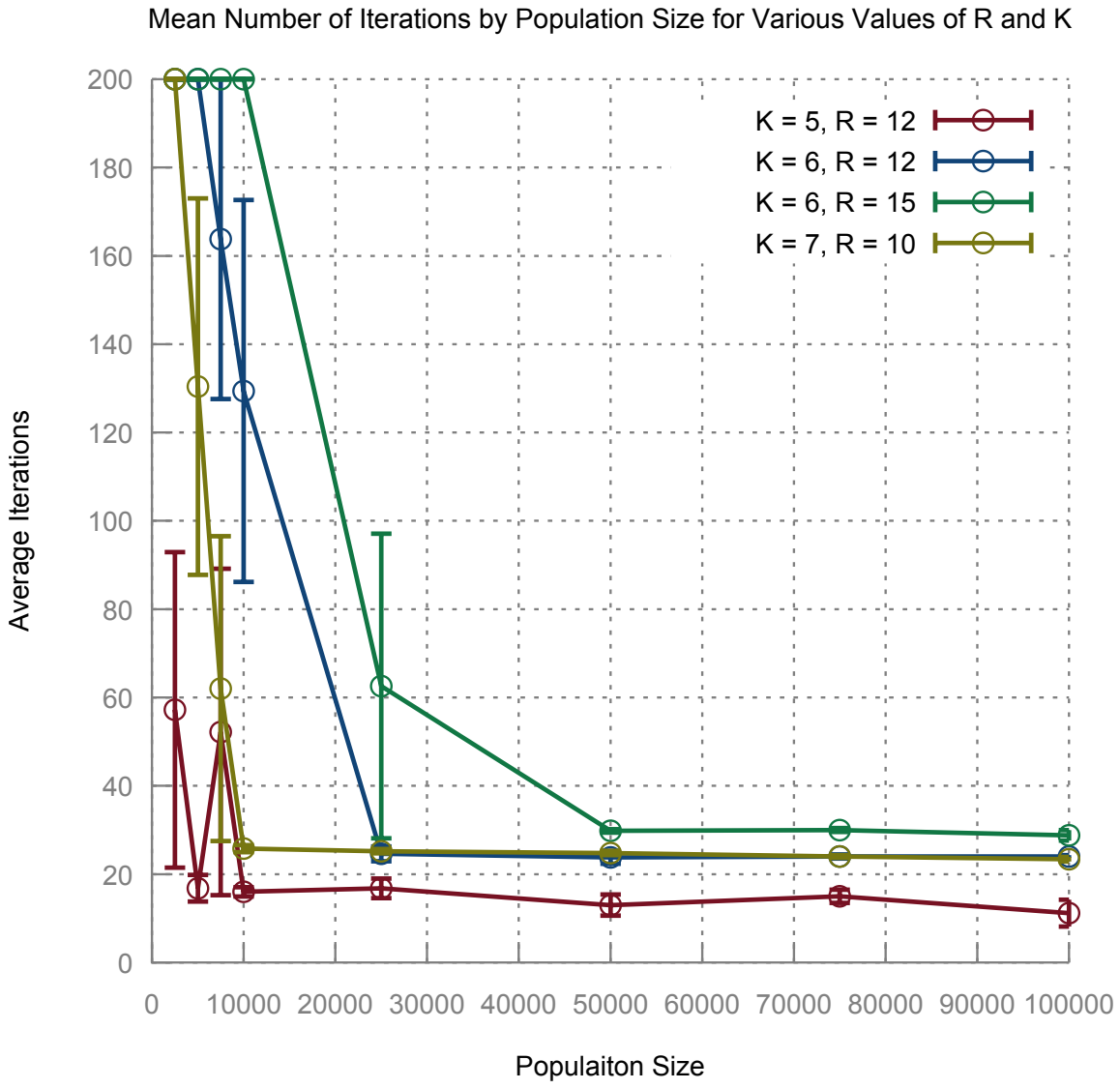


Figure 5.1: Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus Population Sizes for Various Problem Sizes

in a population is changed. One relies on the positive changes being selected for, because the change will result in a greater fitness and an increase in likelihood for selection. Mutation is therefore important in allowing for realistically sized populations to conduct more comprehensive searches.

This section intends to discover what the effect of mutation is on the convergence rate of DePermute. An initial intuition is that with no mutation the convergence rate will be lower for smaller populations, and that some value of mutation will result in the greatest net gain in convergence rate, but that a very high mutation rate will harm the convergence rate by destroying high quality solutions.

Expected Mathematical Relationship
Convergence rate has a parabolic relationship to mutation rate.

Experiment 2 (see below) is designed to test the response of the algorithm to varying mutation rates.

Experiment 2
Trajectory Diversification Experiment:

1. Generate n random problems for increasing problem sizes.
2. Use the results generated by *Experiment 1* to determine a suitable population size for use in each run.
3. Disable sub-space division, block alignments and elitism.
4. Run DePermute three times for each problem, and once for each of five values of mutation from an evenly spaced interval from 0.0 to 1.0.
5. Average the number of iterations taken by the algorithm to reach at least 0.01 fitness (a near perfect solution).

Results

The results indicate that the rate of convergence on a high quality solution is not described by a parabolic function, as was expected. Instead the results illustrate that the number of iterations to find a high quality solution is usually increased by a higher mutation rate. It is concluded that mutation is not a mechanism for increasing convergence rate and ought instead to be used in lieu of sufficient memory – to increase population size – in order to ensure that the entire search space is search-able.

5.2.4 Elitism (Population Refinement)

Elitism is designed to behave as a ratchet, in that it doesn't allow for the top E members of the population to deteriorate, thus the quality of the best solution is monotonically increasing.

Given our intuition of elitism it is pertinent to ask what the actual effect on convergence rates is. It is trivial to note that when all members of the population are elite the population

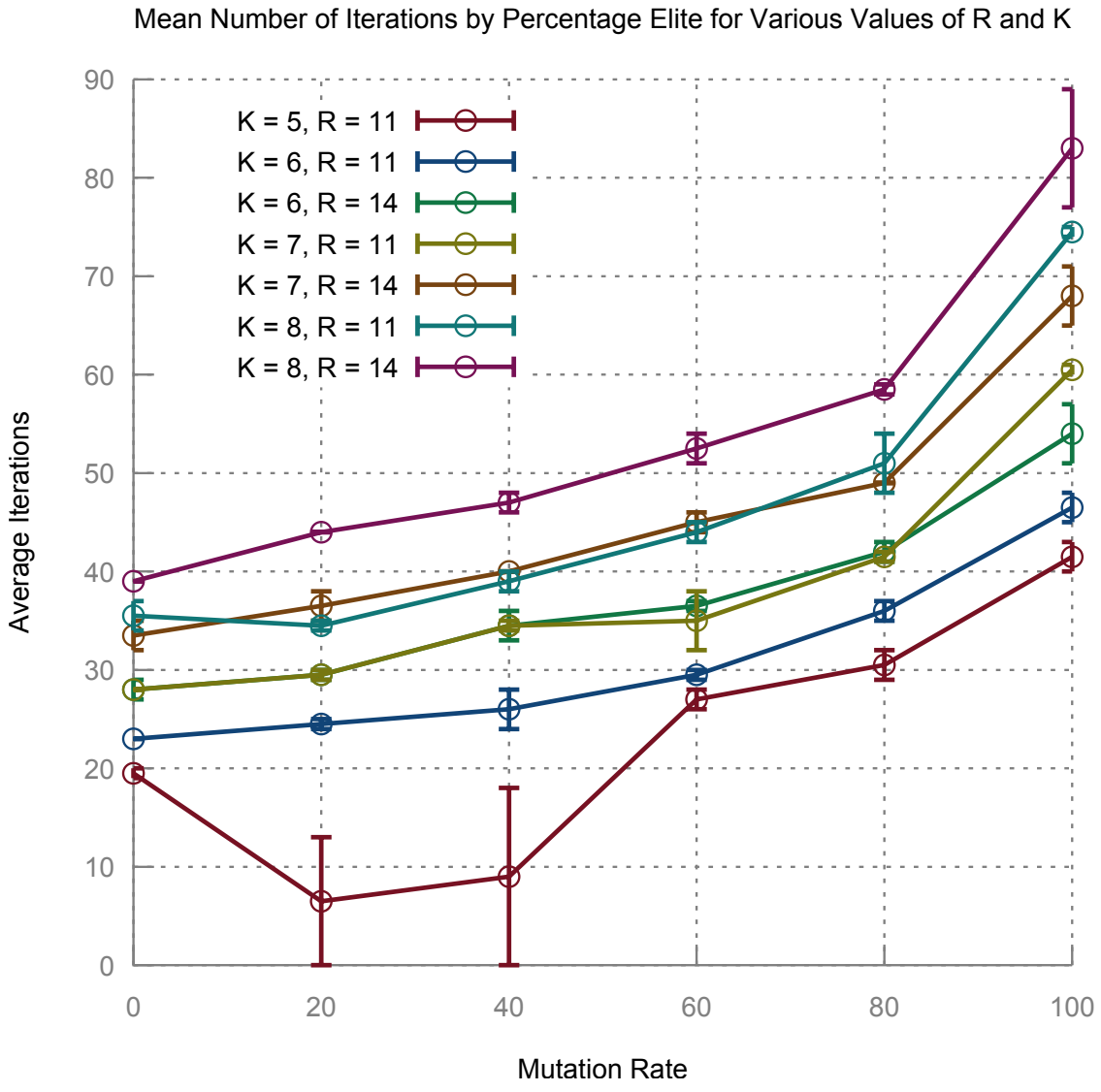


Figure 5.2: Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus Inverse Mutation Rates

will never change and will therefore never improve. When there is no elitism there is a risk that the high quality solutions are destroyed during crossover.

Expected Mathematical Relationship
The convergence rate is positively affected by small elitism values and adversely affected by larger elitism values.

Experiment 3 is designed to test the response of the algorithm to varying elitism rates.

Experiment 3
Population Refinement Experiment:

1. Generate n random problems for increasing problems sizes.
2. Use the results generated by *Experiment 1* to determine a suitable population size.
3. Disable sub-space division, mutation, and block alignment.
4. Run DePermute three times for each problem and once for each of five evenly spaced intervals of increasing percentages of the population in the elite.
5. Average the number of iterations taken by the algorithm to reach at least 0.01 fitness (a near perfect solution).

Results

The results indicate that the expected relationship holds for small K and R but that it does not for larger K and R. Note that, much like mutation, elitism is a practical measure – which has shown effectiveness during ad hoc tests – but in these tests doesn't actually improve the rate of convergence on an high quality solution.

5.2.5 Sub Space Division

The partitioning scheme is designed to force a more thorough search by preventing sub-populations from moving away from areas of the search space prematurely.

Initial Diversification generates random solutions which exist at points distributed throughout the entire search space. Subsequent Refinement operations reduce the amount of the search space being covered and bring the solutions in the population closer to the solutions which lie near optima. Intuitively one can argue that this process is prone to ignoring sections of the search space early.

It is therefore pertinent to ask whether constructing more partitions does indeed have a positive effect on the convergence rate. In order to maintain fairness in the testing procedure, the total number of solutions is kept constant, i.e. each sub-population generates a fraction of the total population in solutions.

Experiment 4 is designed to test the response of the DePermute algorithm to varying the number of partitions which the search space is divided into.

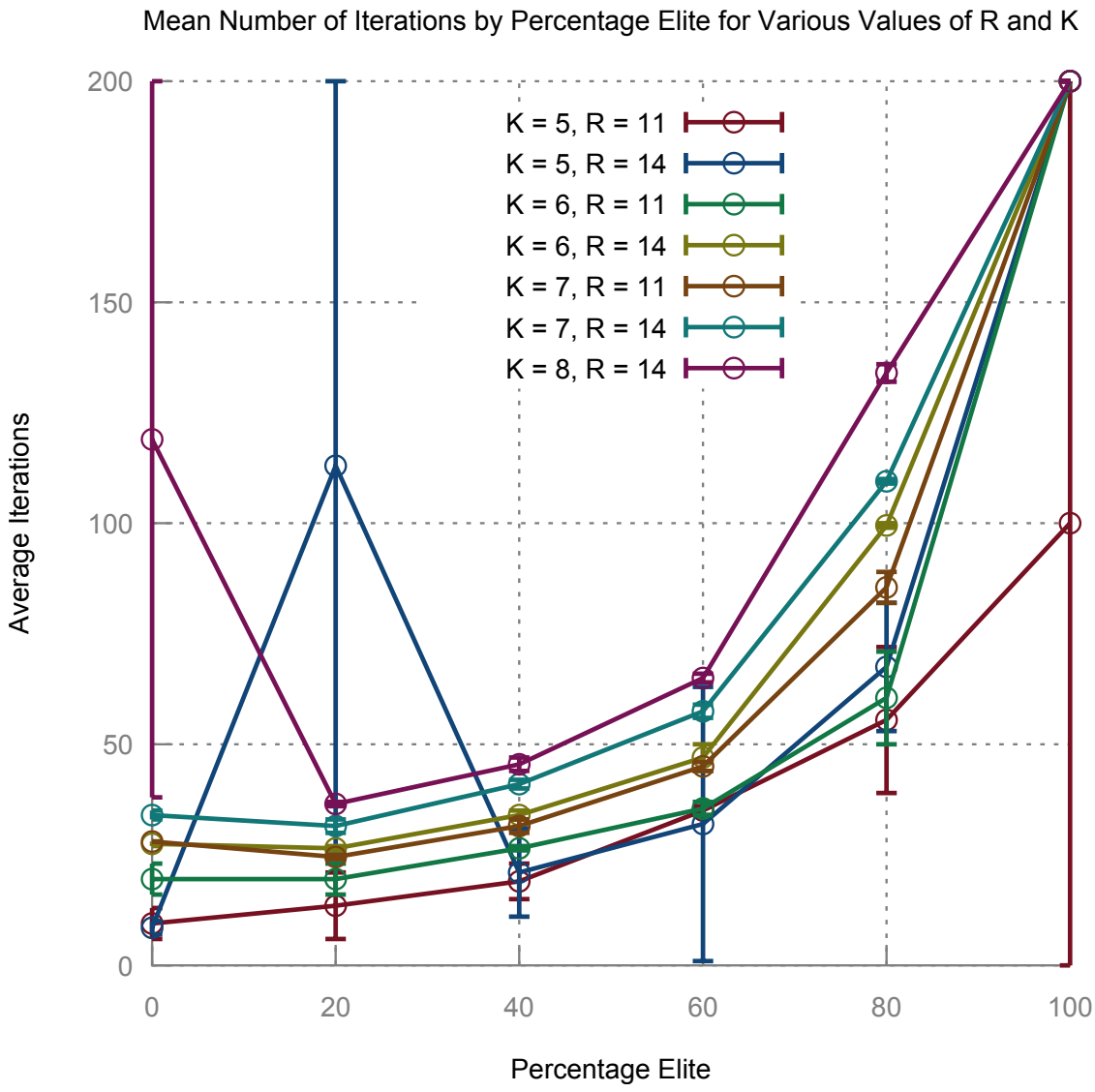


Figure 5.3: Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus Elitist Proportions

Experiment 4
Sub Space Division Experiment:

1. Generate n random problems for increasing problem sizes.
2. Use the results generated in the previous experiments for each problem size to fix the population size, elitist proportion and mutation rate.
3. Disable communication and block alignments.
4. Run DePermute three times for each problem and once with 1 partition, K partitions and K^2 partitions.
5. Average the number of iterations taken by the algorithm to reach at least 0.01 fitness (a near perfect solution).

Results

The results are noisy and a general trend cannot be established. Therefore, it is concluded that no relationship between the degree to which the search space is divided and the convergence rate of the algorithm is apparent for randomly generated data at the given problem sizes. It is further noted that the degree of division was altered in isolation of any communication and it should not come as a surprise that the effect is without a general trend. Division alone is simply running the same experiment multiple times with a smaller population and halting early in the case that one instance of the algorithm finds a solution of sufficiently high quality.

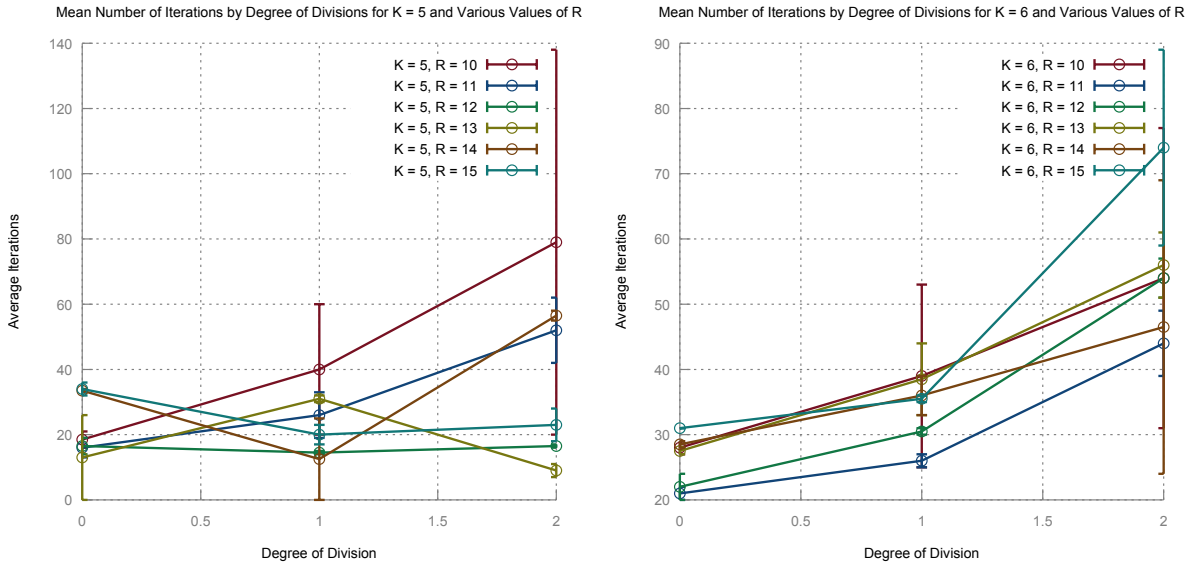
5.2.6 Communication

Sub-partitions of the search space are confined to their own partition for the duration of a run of the program. However, there can be some partitions of the search space, which after a few iterations, could be crossed off with a fair degree of certainty that moving elsewhere would be more promising.

In addition a more thorough search of a promising area, by multiple search entities, might be more likely to find a good answer than one on its own. Communication of the n best solutions is used as an abstract heuristic to support this strategy for searching. If n is equal to the population count in a solver then the searchers will converge faster on an area, whereas if n is zero, the search entities will remain in their partitions.

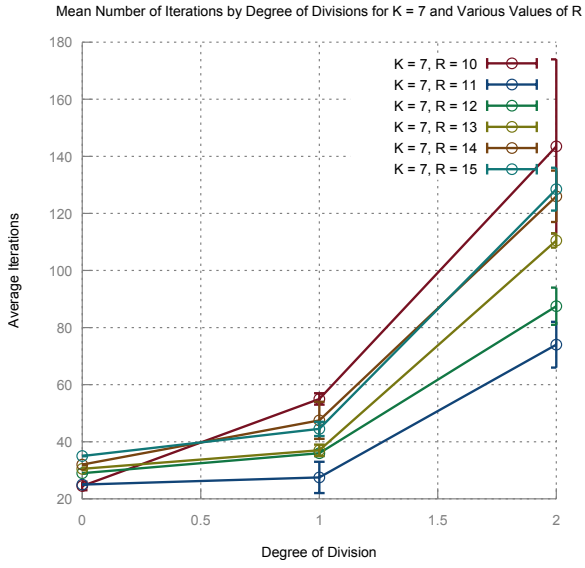
Expected Mathematical Relationship
Communication has a similar effect to that of mutation in that it is ineffective at a value of zero, effective at some point greater than zero and then tapers off, reducing the efficacy of the search as the search entities converge on an area too quickly.

Experiment 5 is designed to test the response of the algorithm to varying degrees of population communication.

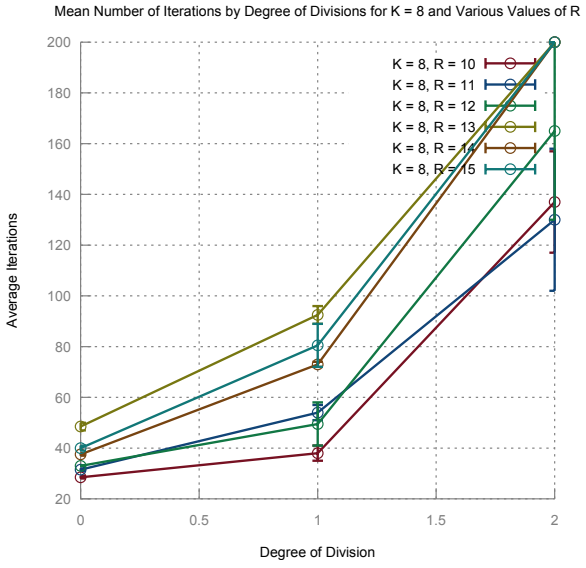


(a)

(b)



(c)



(d)

Figure 5.4: Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus the Number of Divisions Made in the Search Space

Experiment 5
Communication Test:

1. Generate n random problems for increasing problem sizes.
2. Use the results generated in the previous experiments for each problem size to fix the population size, elitist proportion, mutation rate and number of divisions.
3. Run DePermute three times for each problem and once for each of 10 uniformly increasing percentages of the population communicated up to 100%.
4. Average the number of iterations taken by the algorithm to reach at least 0.01 fitness (a near perfect solution).

Results

The results are varied. In some instances there is a significant decrease in the number of iterations to achieve a solution of sufficiently high quality and in others there is a clear increase in the number of iterations. Furthermore, in some instances there is no clear trend what so ever. Therefore, it is concluded that without further experimentation and research one cannot establish any relationship between the percentage communication and the rate of convergence on a sufficiently high quality solution.

5.2.7 Block Alignment

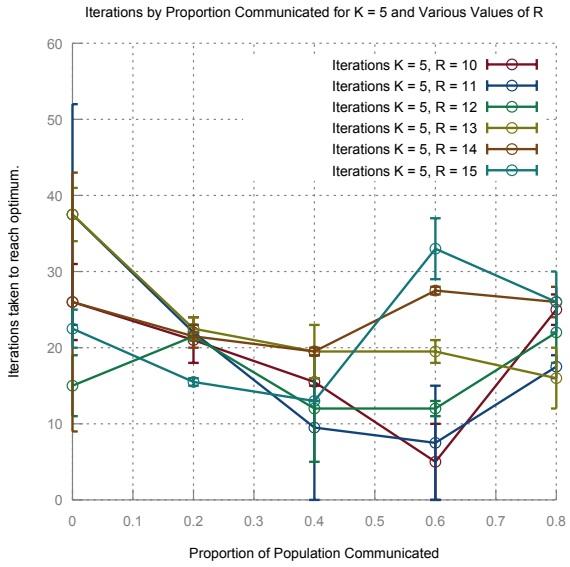
Block alignments, if they exist, reduce the problem complexity dramatically. It is not as important to know how the convergence rate changes as a function of the number of block alignments as it is to know how likely block alignments are found.

Expected Mathematical Relationship
If the number of population groups (columns in the solution) K is low then it is reasonable to expect it to be more likely that a block alignment is found because there are fewer columns to divide one by. In addition increasing M ought to increase the likelihood of finding a block alignment because there are more members in the study and therefore more chances for one to be biased heavily towards one population group.

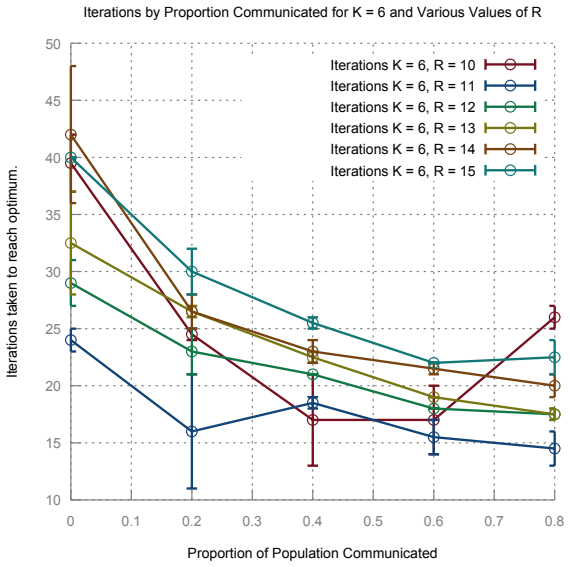
Experiment 6 is designed to establish how likely it is to find a block alignment in different problem sizes.

Results

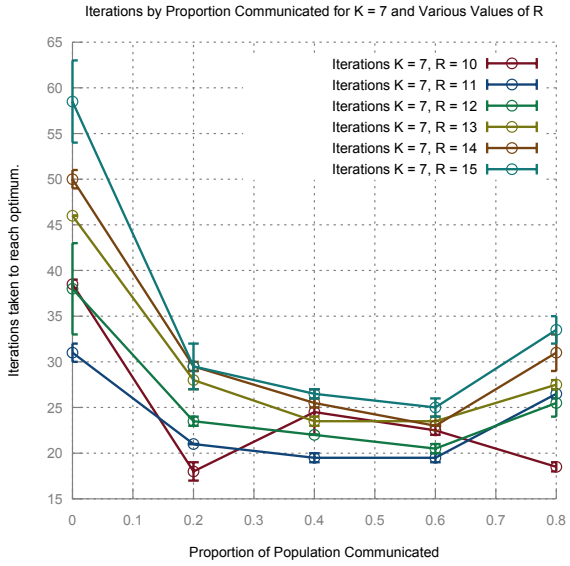
The test results indicate the effectiveness of the block alignments algorithm in a worst case situation. In a practical setting it is entirely likely that marker individuals exist in a population – who belong entirely to one particular population group. In fact we expect to find many block alignments in nature, because of real structure in the data. Each marker individual



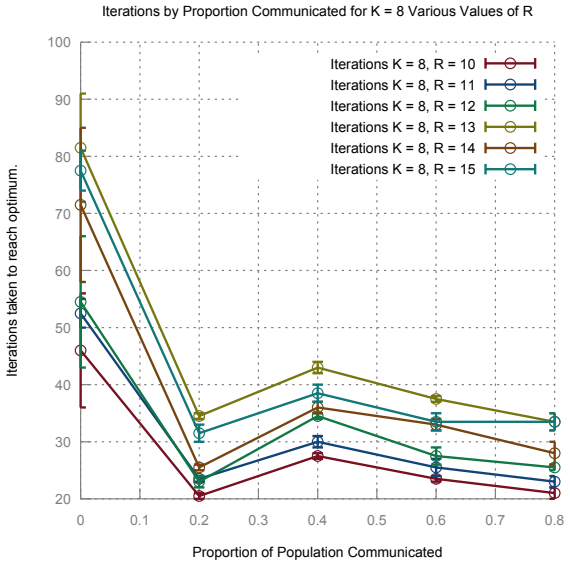
(a)



(b)



(c)



(d)

Figure 5.5: Graphs of the Number of Iterations Taken to Reach an High Quality Solution Versus the Proportion of the Population Communicated

Experiment 6
Block Alignments Test:

1. Generate five sets of n random problems for increasing problem sizes.
2. Run the block alignments stage of DePermute on each of these problems recording the number of block alignments found in each instance.
3. Average the number of alignments found for each problem size.

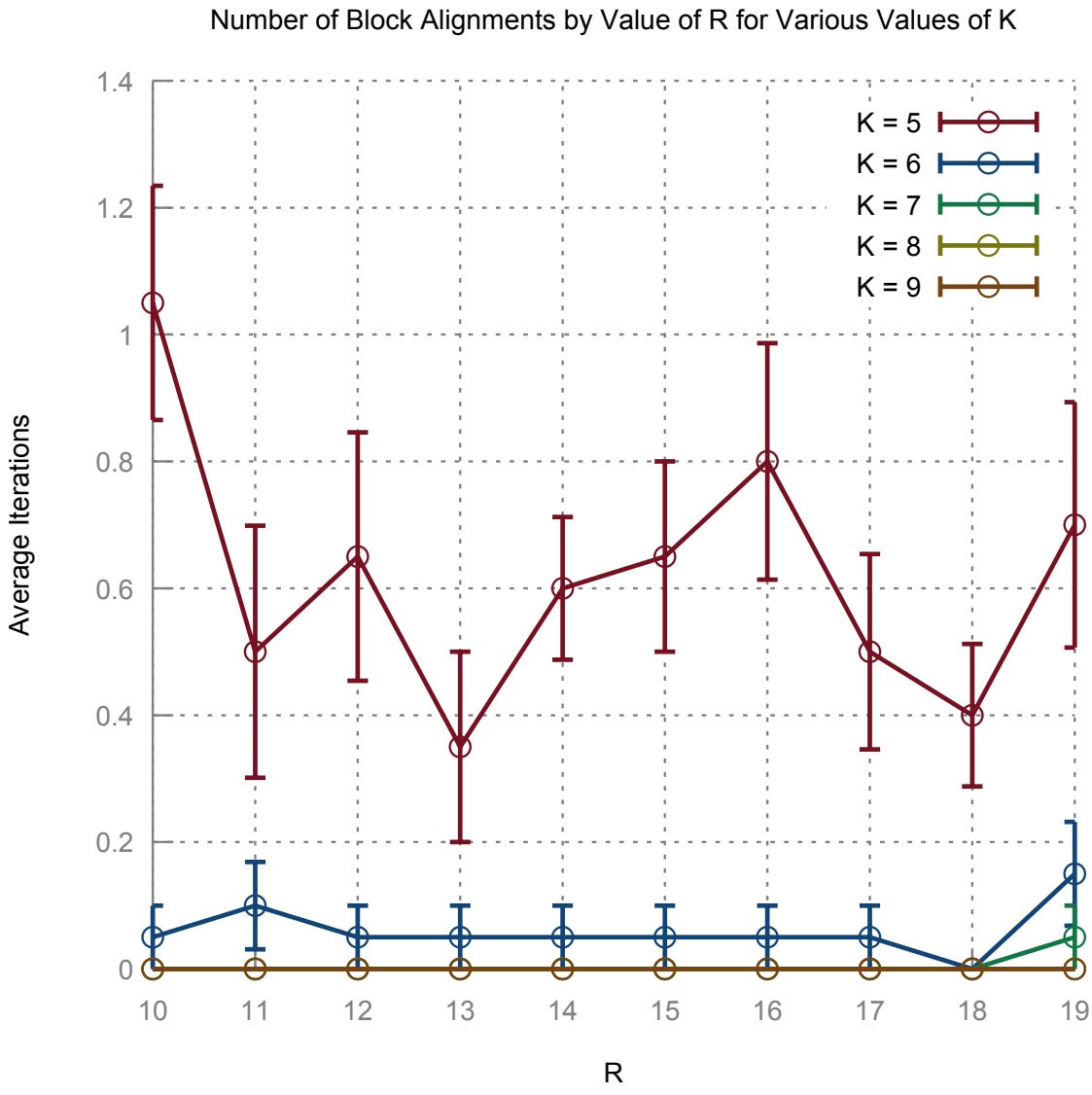


Figure 5.6: Graphs of the Number of Block Alignments Found For Various Problem Sizes

could potentially result in another block alignment. In a randomly generated data set using a uniformly distributed PRNG one would expect for the individuals belongingness to be distributed evenly across population groups. Therefore we expect for a low number of block alignments to be found.

The results indicate a clear relationship between the size of K and the number of block alignments. At low values of K a block alignment is found on each run of the experiment on average. At larger values of K the values in Q matrices are more evenly distributed and block alignments become rarer.

We note that there is no clear relationship between the value of R and the number of block alignments found.

5.2.8 Critical Discussion of Testing Procedure

Exhaustiveness of Testing

The testing procedure adopted in this chapter is not exhaustive. The question as to how the heuristics affect each other is left open. In testing each heuristic we have adopted a greedy optimisation technique by finding optimal values in isolation and keeping them fixed as we find them. For instance, it is possible that the use of finely tuned population sizes reduces the impact of other heuristics used in the genetic algorithm. Furthermore it might be found that the function of convergence rate with respect to the degree of use of a particular heuristic is dependent on the combination of heuristics. i.e. the selection of parameters for DePermute could in itself be a complicated optimisation problem.

Generation of Random Input Data

The second shortcoming of this chapter is the use of entirely random data. In adopting this testing procedure we have elected to ignore the structure which would be present in real data. Alternatively, one could generate data which would more closely mimic the reality of how values are distributed in Q-matrices. e.g. one could randomly select population groups from which an individual was descended and bias their entries in a Q-matrix towards those population groups. Continuing in this way, for other simulated individuals, one could generate the first matrix. Subsequent matrices could be generated using the same biases which were originally generated.

By adopting a random data generation technique which respects the attributes of real data sets one could potentially arrive at conclusions which are more relevant to researchers from a practical perspective. What has instead been discovered is of more theoretical than of practical importance. The use of entirely random data is a worst case input. For random inputs it's likely that the solutions to the problem are very similar. With sufficient testing by this methodology one would be able to validate the computational bounds of the algorithm in the average case.

Chapter 6

Practical Testing

This chapter focuses on the implementations of the DePermute algorithm on a CPU, GPU and FGPA driven computer. The discussion for each prototype is organised into four sections:

1. software design;
2. implementation of algorithm;
3. data structures;
4. optimisations;

C++ is used to implement all host code. It is a flexible language and is well supported for both the GPU and FGPA hybrid computer based implementations. In the case of the GPU based prototype the OpenCL programming language is used due to the availability of equipment at the time of specifying the project. Finally, a combination of Verilog HDL and the x86 assembly language is used to implement the FGPA hybrid computer prototype.

6.1 CPU Based Prototype

The CPU based implementation of the DePermute algorithm was developed for two purposes:

- to be used in convergence based testing;
- to provide a baseline against which CLUMPP is compared and against which the massively parallel versions of DePermute may also be compared;

6.1.1 Software Design

The program was designed using a procedural decomposition strategy. A top-down design approach resulted in fifteen modules. Broadly, these modules can be organised into host code which is responsible for user interaction, search space division and distance caching, and kernel code which performs the iterative computation. This distinction mirrors the structure of programs designed for host/co-processor computers, thus increasing the relative ease of porting the CPU prototype to other platforms.

Figure 6.1 describes the interface to and relationship between the fifteen modules which comprise the program. More abstract modules are arranged toward the top of the diagram and more primitive modules toward the bottom. Note that modules responsible for the i) block alignments ii) Branch and Bound ii) and genetic algorithm have been marked on the diagram.

All other code was implemented using standard libraries and the majority of the implementation was undertaken using standard expressions and control structures. Finally, note that the C++ version 11 language is used because it supports the use of lambda abstraction and type inference (among other features) which was used in non-kernel code for convenience and ease of expression.

6.1.2 Implementation of the DePermute Algorithm

Preprocessing

The code base for preprocessing steps in the DePermute algorithm (including both block alignments and cache computation) is shared across all versions of the program. This is because preprocessing steps are computed once, have low overheads and because it is convenient to implement them in a conventional development environment.

The algorithm to compute the distance cache is a four level nested loop. The algorithm computes and stores the distances corresponding to ADMIXTURE columns according to the following scheme. Compute the distance from each column in Q-matrix i, to each subsequent Q-matrix j (i.e. j always starts from matrix $i + 1$) from each column index m in matrix i to each column index n in matrix j.

The algorithm to compute block alignments is implemented as a simple two-level nested loop through the first Q-matrix. In the instance that a potential hit is encountered then that value is set to the first index in a backtrack array. Next it is determined whether or not the column corresponding to the potential hit has already been marked in the LRT. If it has not then it proceeds to confirm the alignment and find the columns which align.

For each other matrix in the same row a value within a small delta of the potential hit value is searched for. If such a value is not found then the loop escapes and the algorithm continues searching on the next row of the Q-matrix. After traversing all other matrices, provided that the columns which align were determined, the backtrack array is appended to the search space and the process continues searching on the subsequent row of the first Q-matrix.

Branch and bound is implemented using two vectors of "parent" data structures – "current" and "children." See *Section 6.1.3* for a description of the parent data type. Initially the current vector contains the parent which corresponds to the un-partitioned search space. For each parent in the current vector a collection of up to K parents are added to the next vector – corresponding to as many partitions of the corresponding parent. Parents are filtered from the queue according to the bounding algorithm discussed in *Section 6.1.3*. The process continues while the size of the current vector is less than the number of partitions which were requested.

Parallel Solvers

The discussion will follow the chronological order of operations in the genetic algorithm implementation as illustrated in *Figures 6.2 and 6.3*. *Figures 6.2 and 6.3* depict the CPU based

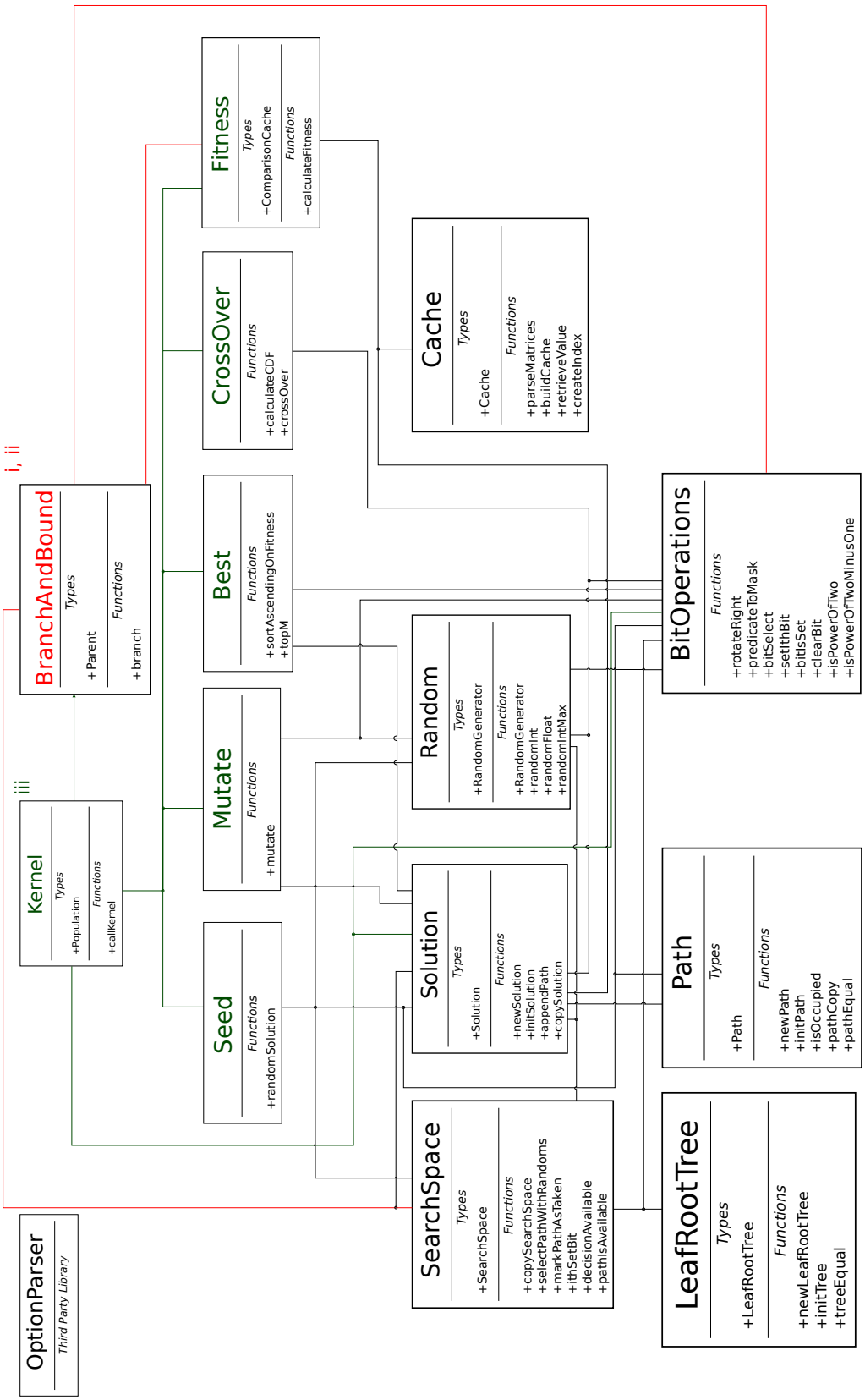


Figure 6.1: Module diagram for the CPU DePermute program

implementation of the algorithm at a high level and can be read in a similar way to a flow chart.

First the program generates a starting population using the depicted three step process. Next the fitness of each solution is evaluated using the lookups defined by the position of arrows which loop through positions in the matrix on the left side. The entire population is sorted on fitness in the next part of the figure, following which termination of the program is flagged to all threads if a solution of sufficiently low fitness value is found. The next part of the figure depicts both the communication of the highest quality solutions to neighbours and the selection of two solutions to perform crossover on. Finally mutation is depicted as the selection of a random number which, if sufficiently low, triggers a Knuth shuffle on a random row of the current solution matrix.

Details In previous sections an overview of the process described by *Figures 6.2 & 6.3* was presented. The discussion now moves to a more detailed description of the important aspects highlighted in the diagram.

First a buffer of buffers of solutions is populated. This structure acts as the amalgamation of all populations and duplicates as the method for communication between solvers.

Population seeding works in constant space. First a copy of the original search space is constructed. The copy can then be mutated while the original is used as a template for the generation of further random solutions. Then random nodes from those remaining in the LRT are added to the partial solution and crossed off from the LRT until no more nodes remain in the LRT. This simple algorithm is analogous to Knuth Shuffling and therefore constructs fair, uniformly distributed random solutions.

Fitness calculation is the first step of the iterative procedure. It is carried out by a four level nested loop, first iterating over all solutions, then over each column, and within each column from each index i to each subsequent index j fetching the cached distances computed for the column values at i and j corresponding to the rows they are in and summing these to a running total – see *Section 6.1.3* for a description of how values are retrieved. In the case of the fast fitness function (see *Equation 4.2*) the index i is set to zero so that only the first row of each column is compared to each subsequent row. Finally the average is taken by dividing by the total number of lookups.

Next the population is sorted on fitness. Sorting is accomplished using a simple bottom up merge sort. A custom sort was required to sort key value pairs which were in separate buffers, bottom up merge sort was selected for its simplicity. In addition $O(n \lg n)$ is much smaller than the complexity of fitness evaluation, therefore any major improvements to this routine wouldn't change the overall running time by a significant amount.

From the sorted population it is possible to compute whether a solution is within the cut-off threshold. If such a solution exists, then it is copied into a shared buffer and the thread which found the solution then terminates. If a solution within the cut off is not found then each thread inspects the shared buffer to determine whether another thread found a solution, if a solution is found there, then that thread also exits.

Once the population has been sorted, solutions can be communicated. Solutions are copied into the lower end of their neighbouring solver's buffer. In order to make this protocol robust it would be required that threads synchronise once per iteration, but synchronisation can incur severe overhead penalties.

The communication protocol for the CPU prototype is not completely robust. One solver could write into its neighbour's buffer before its neighbour's buffer is sorted, thus risking overwriting the best solutions in that buffer. Furthermore note that the neighbour buffer might

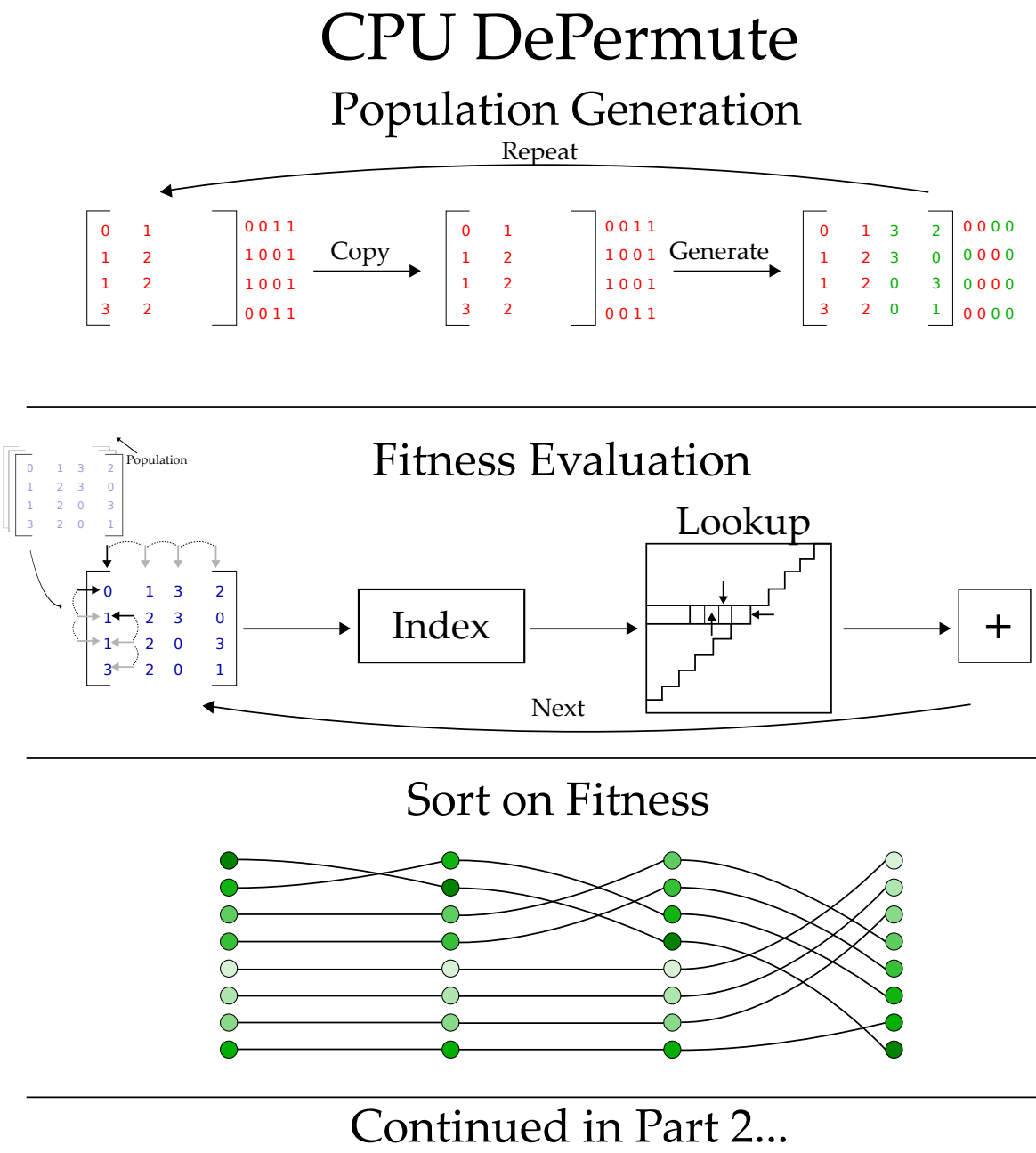


Figure 6.2: Outline of the DePermute algorithm for the CPU – part 1 of 2.

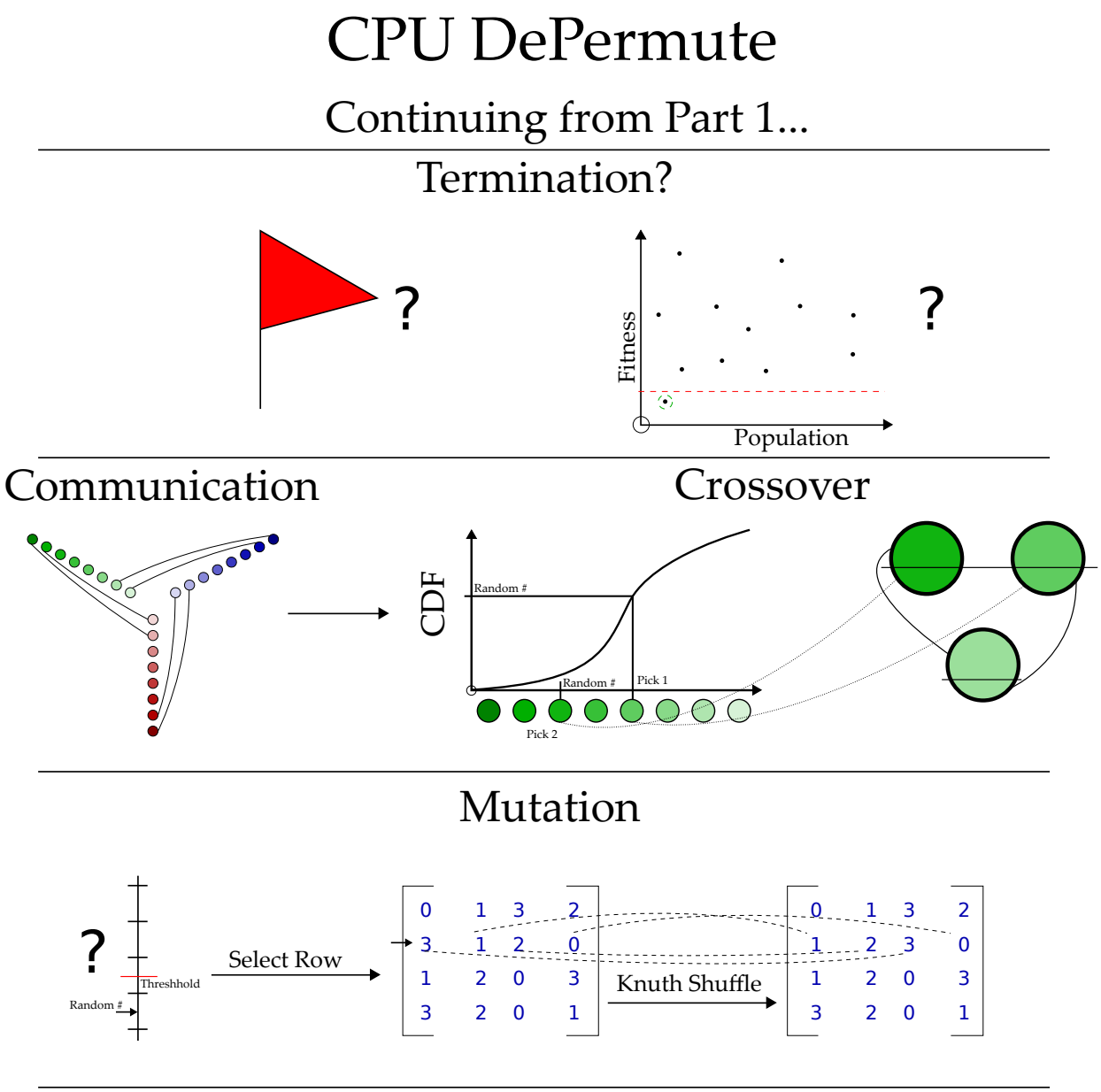


Figure 6.3: Outline of the DePermute algorithm for the CPU – part 2 of 2.

not contain the current generation because crossover writes into a scratch buffer and when an iteration completes the scratch population buffer is swapped for the current population buffer.

It was decided that the performance benefits of asynchronous operation are more valuable than the benefits of complete robustness. Finally, note that mechanisms have been established in the fitness function to punish invalid solutions in order to recover from erroneous behaviour and make it unlikely that such solutions are selected for.

The Cumulative Distribution Function (CDF) is evaluated by a simple two-pass loop. On the first iteration an accumulator is set to zero. The sum of the accumulator and the next element is then added to each index. By the end of the first pass the accumulator has the value of the sum of all fitnesses. Each element can be divided by the accumulator in a second pass so as to normalise the values.

Elitism is bundled into the crossover operation, whereby the top solutions are copied into a "next generation" buffer. Following which, the crossover step continues using the selection of two solutions by the following process:

- A random number is generated and the solution in the CDF corresponding to the random number is selected.
- A solution is selected at random.

Selection under CDF is accomplished by binary search (since a CDF is strictly increasing). An iterative implementation of binary search is selected for this purpose. Following the selection of two solutions a solution in the next generation is constructed by copying a portion from the top of the first solution and a portion from the bottom of the second solution into the next solution. The proportion of the copy is determined by random selection of a row in the solution matrix. This strategy is often referred to as single point crossover.

If recombination is enabled then the procedure is undertaken during the crossover stage of the algorithm. Recombination simply computes a random recombination point in a select few solutions and swaps the top portion of the solution with the bottom portion.

Mutation is undertaken by a simple loop through all solutions in the population. A random number is generated to determine whether the solution ought to be mutated. If that solution is selected for mutation then a row is selected at random and shuffled.

At the end of an iteration the current generation pointer is swapped for the next generation pointer and the process repeats itself.

6.1.3 Data Structures

Preprocessing

There is an inherent trade off in the choice of storage scheme for Q-matrices when computing block alignments when compared to the process for computing the distance cache. Distance cache computation iterates through the columns in ADMIXTURE matrices to compute F_{norm} between all pairs of *columns*. The block alignments algorithm iterates through all the elements of the first matrix and, if it finds a hit, all the elements of the *same row* in all other matrices.

The former procedure is best supported by a column major format of the matrices because the columns can be completely or partially stored in cache pages. The latter is best supported by a row major format of the matrices for the same reason.

The relative frequency of row versus column-traversals is used to decide which format is most effective. In the case of computing the cache, it can be seen that the entire cache must be computed regardless of the structure in the data. It can be shown that there is a maximum of one block alignment corresponding to each row (provided that the threshold for the heuristic is greater than 50% the remaining values can't be flagged because they must be $< 50\%$) therefore we are guaranteed to make fewer row traversals than we are column traversals. According to this reasoning it is clear that the admixture matrices ought to be stored in column major format.

An alternative approach to selecting a matrix format on the basis of the most frequent way it is traversed is to instead transform between formats so that the matrix is always in the most appropriate format for a particular stage of the algorithm. The cost of this transformation across all matrices is linear in the number of elements stored. The complexity is therefore equal, in asymptotic complexity, to the cost of the block alignments algorithm itself. Testing whether this approach would provide a speedup which would warrant its use is outside of the scope of this dissertation [1].

The "parent" data structure is used to abstract the details associated with memory management when storing search space data structures in the C++ vector container. The data structure contains a Leaf Root Tree and a Solution and provides methods to copy itself, construct itself and free its own memory. This allows for the correct use of the standard vector library without memory leaks.

The selection of a representation for the LRT data structure involves a time/space trade off. Marking a node as taken requires one bit. This information can be represented by a buffer of boolean values, however the native representation for a boolean value is eight bits wide. This representation is wasteful. We can instead decide to represent a boolean value buffer using a collection of bytes and write into the bit corresponding to the index requested. However this involves overhead in computing the byte in which the bit resides and the bits index within the byte.

Instead a collection of paths taken using is represented by an integer. This restricts the total number of paths to either 32 or 64, depending on compiler and platform architecture, but eliminates the need to compute the byte in which the bit resides. The latter representation is space efficient and time efficient, but is limited in problem size. However, a population study with more than 32 population groups has not been found in the literature, therefore a 32 bit integer should be sufficient in most applications. Finally, we define the LRT as a buffer of integers, to represent the nodes on each row.

The column distance cache is represented by a flat array which is indexed in four dimensions. The function for indexing the cache follows from the four level nested loop which builds it, and is given in *Equation 6.1*, where K and R are the dimensionality of the solution and i and j are the column indices of columns in admixture matrices n and m respectively.

$$-\frac{(K^2 i(i-2R+1))}{2} + (j-i-1)K^2 + mK + n; \tag{6.1}$$

Two alternative representations are also analysed elucidating the penalty in storage which each incurs.

A high dimensional matrix could have been used for the representation. However, column distance is commutative, therefore the cache need only represent the distance of columns

[1] My intuition is that it would indeed be more effective than selecting one format.

corresponding to values which occupy a higher row in the solution to those which are on subsequent rows.

A flat C++ style matrix is not capable of having different lengths of sub arrays and would therefore use memory proportional to *Equation 6.2* instead of *Equation 6.3*.

$$\sum_{i=0}^{K-1}\sum_{j=0}^{R-1}\sum_{k=0}^{R-2} 1 = KR(R-1) \tag{6.2}$$

$$\sum_{i=0}^{K}\sum_{j=0}^{R-1}\sum_{k=j+1}^{R-2} 1 = \frac{R(K+1)(R-3)}{2} \tag{6.3}$$

If instead pointers are used to store different length arrays for subsequent rows in the cache then we pay the price of storing the pointers. Inspection of the storage requirement in *Equation 6.3* shows that only the inner-most dimension contains arrays which have shorter lengths (note the increasing starting values of the sum variable k). Therefore a pointer based representation would waste space proportional to *Equation 6.4*.

$$\sum_{i=0}^{K}\sum_{j=0}^{R-1} 1 = R(K+1) \tag{6.4}$$

Finally, the rows in a pointer based representation are likely not to be local to each other in memory, because the rows are dynamically allocated. Therefore the pointer based representation could make poor use of the CPU cache.

Parallel Solvers

A solution is represented in column major format, as a collection of paths through the LRT, where a path through the LRT is a column of the solution matrix.

Instead of using a flat array representation a collection of pointers to paths is used. Indexing the latter structure requires two pointer look ups in addition to a read operation for the corresponding index. The alternative flat array representation involves a multiplication, addition and then a pointer look up.

The former representation can lead to bad spatial properties for cache paging, but is easier to implement and it is difficult to predict whether the performance cost is negligible or not.

A population of solutions is represented using a buffer of pointers to solutions. This allows for two threads to modify spatially local solutions without changing the population buffer, thus minimising cache thrashing from cache snooping protocols.

Additionally the solutions can be sorted without copying solutions during swaps by instead swapping pointer values. In this instance the pointer to a solution is used in the same way as one would a position key.

Finally consider the structure of data used to represent the communications channel. Each solver thread has direct access to the initial population data structure of the other threads by a buffer of pointers to each solver's initial population. This supports communication by writing into a solver's neighbouring buffer. A thread can determine which population to write into by adding one, modulo the number of solvers, to its thread ID and dereferencing that pointer to access its neighbouring population.

6.1.4 Optimisations

Parallel Solvers

Allowing each thread to construct its own resources (which other threads cannot access or mutate) is important in avoiding cache thrashing, because multiple writes to spatially local data by multiple processors would incur costs in the cache snooping protocol of the CPU.

A master procedure spawns a C++11 thread for each of the search space divisions provided to it (as parent data structures from the Branch and Bound process). C++11 threads are implementation specific. For project testing the GCC is used, thus the POSIX thread library is used as the underlying implementation.

Each thread acts on its own population and does not synchronise to other threads. Aside from the population, each thread owns a buffer of floating point values corresponding to fitness values, along with a scratch buffer for temporary storage, a buffer of floating point values corresponding to the CDF and a scratch buffer containing solutions which acts as the subsequent generation of solutions. The lack of synchronisation leads to race conditions in the writing of solutions into neighbouring populations. This could cause populations to become corrupted with invalid solutions; however the fitness evaluation step provides a simple mechanism to penalise any solutions which are invalid. The high penalty for corruption makes it very likely that corrupted solutions are removed from the population during selection.

Each thread also constructs four random number generators, three for performing crossover and one for performing mutation. Note that three random numbers must be generated for each crossover (one for each solution and one for the crossover point).

If one pseudo-random number generator (PRNG) is used for crossover and one for mutation then the period of the PRNG for crossover would expire up to three times faster than the PRNG for mutation because three numbers are generated for each crossover operation and on average one number is generated for a mutation operation. An alternative would be to use a single PRNG for all numbers, but its period would expire four times faster than the use of four separate generators. In addition, if one PRNG is consumed by multiple sources there is no guarantee that the sequence of numbers which each source receives has the same properties as it would had it had a dedicated PRNG. i.e.if a PRNG algorithm guarantees that it generates numbers with an even distribution then there is no guarantee that the same will hold when only accepting every n^{th} number which it generates. The decision was taken to use multiple distinct generators in order to support longer running times and because of the requirement that there is sufficient entropy for a stochastic search to be conducted.

All implementations of the DePermute algorithm use the "xorshift PRNG" algorithm. The xorshift PRNG algorithm has a long period and is well distributed [4, 35]. It is also arguably faster than the well known "Mersenne Twister" PRNG algorithm [4, 35]. It is implemented as listed in *Listing 6.1.* Note that the state of the random generator `xs` is seeded with "true" random data (implemented on UNIX style systems by reading from `/dev/random`) before use.

Ad hoc testing during development of the program also showed that the use of a single PRNG resulted in unsatisfactory behaviour of the program. The population quickly became stratified at various local minima early on in the search when a single PRNG was used throughout the program. Stratification was not observed when multiple PRNGs were used in the program. An explanation of this phenomenon is outside of the scope of this dissertation, but a possible explanation is that the phenomenon relates to the random numbers becoming correlated when there is insufficient entropy in their generation.

Listing 6.1: XORshift PRNG algorithm as used in all implementations of DePermute.

```
// Given:
// l           % A pointer into the state of the PRNG
// SEED_LENGTH % The length of the seed buffer (note that this impacts the
    period of the PRNG)
// xs          % The state of the PRNG


al = (l + 1) % SEED_LENGTH;
a = xs[al];
a ^= a << 15;
a ^= (a >> 14);

bl = (l + 2) % SEED_LENGTH;
b = xs[bl];
b ^= b << 12;
b ^= (b >> 17);

xs[l] = a + b;
```

6.2 GPU Prototype

The GPU based implementation of the DePermute tests the efficacy of the algorithm in a massively parallel context. The GPU has arguably become the most ubiquitous massively parallel device available to the general consumer and to the specialist. Therefore, it is pertinent to ask how effective the algorithm is on this device.

6.2.1 Software Design

For the most part the host code for the GPU prototype is shared with the CPU prototype. A great deal of code is added in the setup and initialisation of the OpenCL compute environment. This involves the creation of:

- objects which manage communication with the device;
- buffers to communicate search spaces to the device for initial population generation;
- buffers which only exist on the device to store populations, fitnesses etc.;
- kernel objects and source files for execution on the device;

In order to save time when the program is run multiple times, kernel sources, which are compiled at run time, can be cached by the OpenCL implementation. The co-processor code is arranged into a module per step in the genetic algorithm. These include modules for each of the following steps:

- initial population generation;

- fitness evaluation;
- sorting;
- communication;
- cdf computation;
- crossover;
- mutation;

After compiling each of these modules and executing the population generation kernel, the main program enters a loop in which each kernel is called in turn to form a single iteration. The same rules of escaping and cut off iterations apply to this process as do the CPU and FGPA hybrid computer implementations.

Another change to the structure of the host program is the addition of objects which build options for the program. These parse user option selections into macro definitions. When each kernel is compiled constants and calculated quantities throughout the kernel are injected as macros.

Figure 6.4 illustrates the relationship between the modules present in the GPU based implementation of DePermute.

OpenCL does not support C-style modularity as it cannot perform linking. In order to circumvent this limitation, separate files are included by preprocessing directives and must be programmed as though there is one file (the file which includes the module). Therefore, care was taken not to pollute the name space, and to disallow dependency cycles among modules, because both of these issues result in compile time errors.

6.2.2 Implementation of the DePermute Algorithm

Parallel Solvers

The discussion will follow the chronological order of operations in the GPU implementation of the genetic algorithm as illustrated in *Figures 6.5 & 6.6*. *Figures 6.5 & 6.6* can be read in the same way as *Figures 6.2 & 6.3*, except that some of the steps have been referenced instead of being included in the figures.

In *Figure 6.5*, a starting population is generated in parallel with threads. This is depicted by circles, which represent threads, and solid arrows indicating the location where the threads perform their work. Dashed arrows indicate where the threads move to on the next iteration. The next part of the figure indicates how the order array is initialised at the start of each iteration using an effective GPU style parallelisation of the task. The next part of the figure depicts the process for evaluating fitnesses, which is two-part:

1. The values in each solution matrix are transformed to values corresponding to partial fitnesses.
2. Each solution matrix is interpreted as an array and summed in parallel.

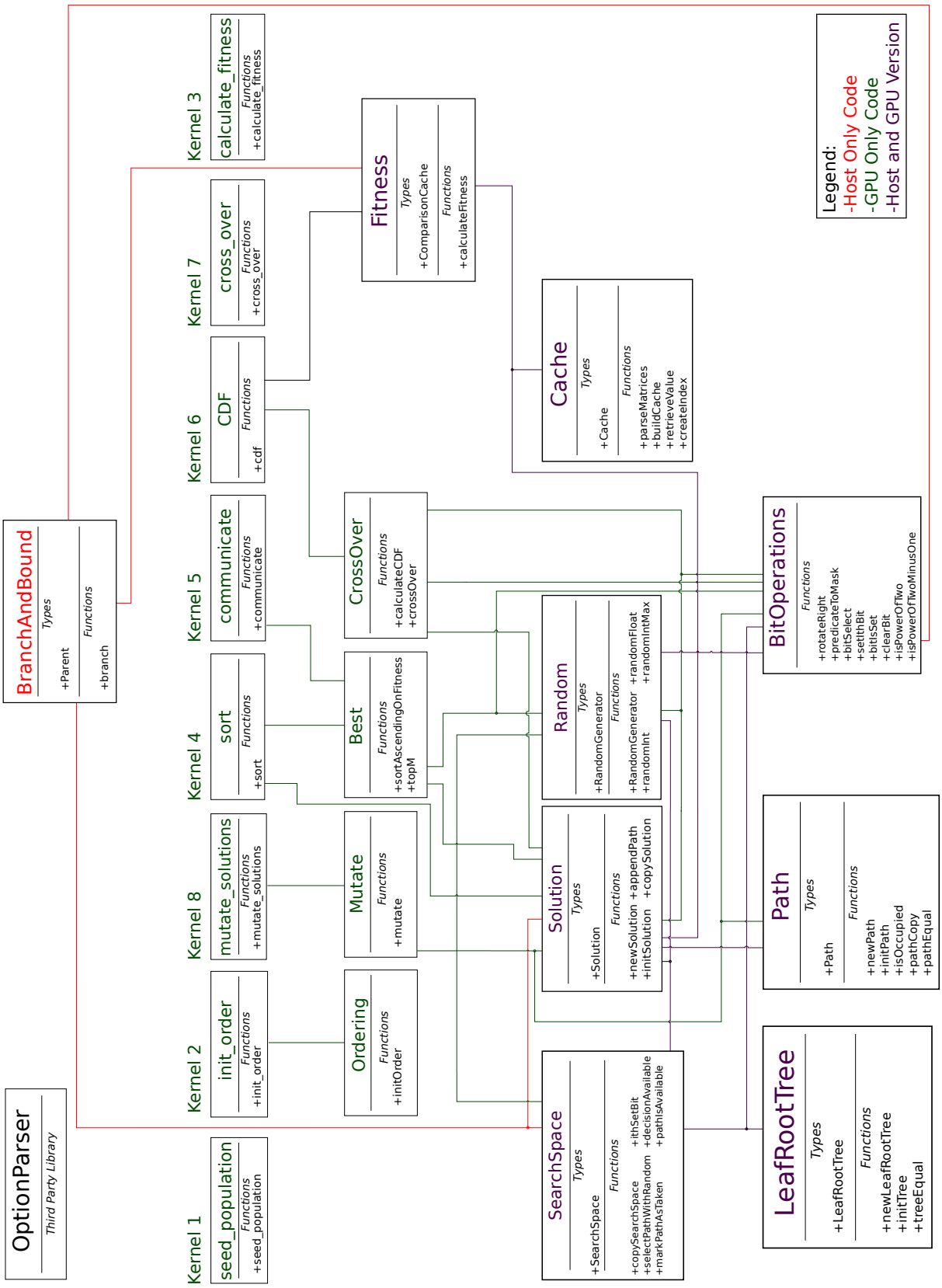


Figure 6.4: Modular decomposition of the GPU DePermute program

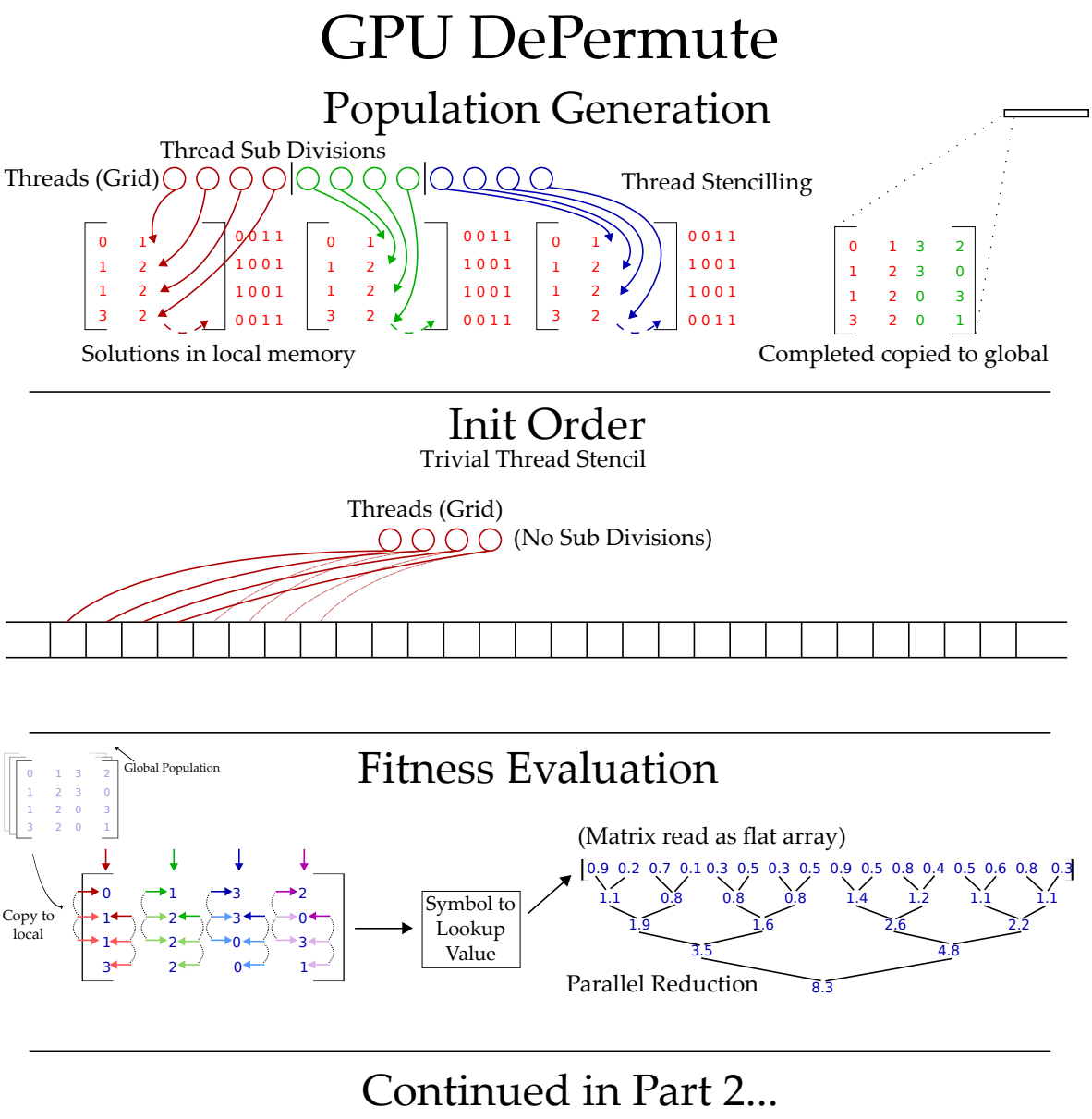


Figure 6.5: Illustration of the implementation of the DePermute algorithm for the GPU – part 1 of 2.

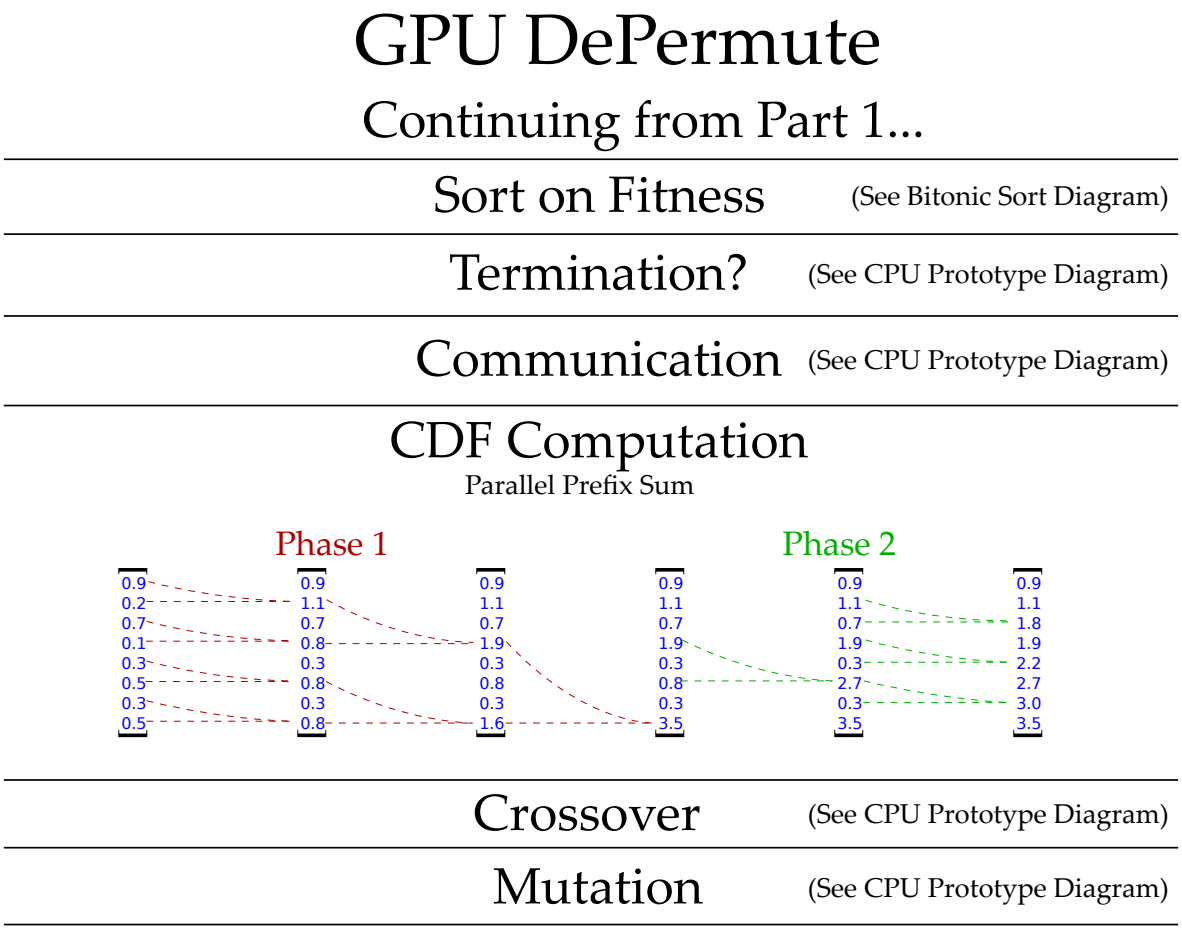


Figure 6.6: Illustration of the implementation of the DePermute algorithm for the GPU – part 2 of 2.

Listing 6.2: Sequential initialisation of a path on the CPU based prototype

```
for ( unsigned int l = 0; l < R; ++l )
    path[l] = -1;
```

Listing 6.3: Parallel initialisation of a path on the GPU based prototype

```
for ( unsigned int l = offset; l < R; l += incrementer )
    path[l] = -1
```

Figure 6.6 depicts the creation of a CDF using a parallel prefix sum with two stages.

Population Generation

A parameter of the GPU population seeding kernel is the number of seeds which are processed in parallel. This is computed as the highest power of two below the number of solutions which can fit in GPU local memory, i.e. the memory shared between PEs in an SM.

Threads are divided between each of the local solutions and perform the same loops as are used in seeding the CPU version of the program – except that threads stencil themselves across the indices considered by the loop. For instance, consider the CPU method to initialise a path given in *Listing 6.2* and contrast it with the GPU version given in *Listing 6.3*

Init. Order

In the GPU version of the program each thread computes an offset within its group as its "local id" by dividing its id by the number of parallel solutions being computed. For the most part the GPU implementation contains nothing more than trivial parallelism, as was the case in the seeding process.

Fitness Evaluation

Fitness calculation is arranged into two stages:

- The symbols in each solution are set to a partial fitness corresponding to the inner-most loop of the CPU based implementation.

  this is undertaken on a row-by-row basis, i.e. threads are blocked from continuing to the next row while a row is being computed;

- The values stored in the solution are summed using a parallel reduction.

  the parallel reduction is $O(\log_2(KR))$ whereas the computation of partial sums is $O(\frac{R(R-1)}{2}K)$;

Parallel reduction is not significantly faster than the CPU version because the sums in the CPU version can be computed as columns are completed, however it does allow for fast aggregation of results. A thread is used for each of K columns per row. The resulting algorithm is bounded by $O(\frac{R(R-1)}{2})$, provided that there are at least K threads. This computation can however become bound by memory because the primary unit of computation in the algorithm is a memory read.

Sort on Fitness

The structure of solutions on the GPU is different from that used on the CPU because OpenCL does not support dynamic memory allocation. In lieu of the language features needed to implement pointer based sorting, a key array is constructed. The key array stores the current ordering of solutions in the population.

At the start of each iteration of the GPU prototype, the key array needs to be reinitialised to indicate that it is referring to the next generation. Solutions are labelled $0 \ldots (\textit{population size} - 1)$. A sorting algorithm on a GPU need be designed for parallelism. Sequential sorting algorithms (such as quick sort and merge sort) exhibit data dependencies which would force a GPU based implementation to serialise much of the algorithm. A comparison based sorting algorithm which is suitable for this purpose is the bitonic sort algorithm [54].

The implementation of Bitonic sort used in the GPU prototype as well as the details of the algorithm are discussed in *Appendix F*.

CDF Computation

GPU DePermute computes the CDF using a parallel prefix sum algorithm. The algorithm works in two stages. First a block of the most fitness values as can be copied to local memory are copied into local memory. The parallel prefix sum for each block is then calculated using a standard parallel prefix sum algorithm [11].

To compensate for the fact that a number of blocks could have been computed prior to the current block, at any point in the algorithm, the sum of the values in the current block is always stored in the zeroth index of the block when work on the current block is complete.

Further details on the implementation of the algorithm may be found in *Appendix F*.

Crossover and Mutation

Mutation and crossover are embarrassingly parallel, i.e. all computations are independent of each other. Once again threads are stencilled across the population and perform the crossover and mutation algorithms sequentially on their allotted indices.

6.2.3 Data Structures

Parallel Solvers

The OpenCL compute platform does not support all the features of the C and C++ languages. In particular OpenCL does not support pointers and dynamic memory allocation. Therefore, the matrix structures used to represent solutions in prior sections must be altered.

The entire population for the GPU prototype is allocated as one contiguous array. An indexing scheme for the array is given in *Equation 6.7*.

$$\textit{indexPop}(i) = \textit{populations}[i * K * R * P] \tag{6.5}$$

$$\textit{indexSol}(i) = \textit{population}[i * K * R] \tag{6.6}$$

$$\textit{indexVal}(i) = \textit{population}[i/(K * R) + i\%(K * R)] \tag{6.7}$$

Indexing this data structure for particular values can become costly on a GPU because integer division and the modulo operator have been known to take an order of magnitude, or more, longer than ordinary operators.

6.2.4 Optimisations

Parallel Solvers

GPU kernels are compiled during execution of the host program. This has allowed for a simple, powerful optimisation to be implemented on the GPU prototype. All parameters are defined via compile time macros supplied by the host program to the OpenCL compiler. This allows for the compiler to compute many of the simple offsets and constants, which depend on user input, throughout the GPU program prior to run time. The work allocation scheme can also be tuned to match the requirements of algorithms. This is because the thread count can also be supplied to the program prior to run time.

There are a number of assumptions which limit the number of threads. For instance there cannot be too many or too few threads and in the case of computing fitness; there cannot be fewer threads than K. Otherwise, the maximum number of threads is limited by the number of parallel threads which the device can support.

6.3 FPGA Hybrid Computer Prototype

The FGPA hybrid computer-based implementation of DePermute tests the efficacy of the algorithm in the context of FPGAs for their ability to cut through paradigms and provide very low level control of synchronisation in a parallel context where a CPU or GPU simply cannot provide the same implements.

6.3.1 Software Design

The design philosophy for DePermute on the FGPA hybrid computer is significantly different from that of the GPU and the CPU prototype. Instead of implementing all parts of the program (which might work effectively on the co-processor) a hot spot in the original program is identified and targeted for acceleration with application specific hardware. This results in two major changes to the CPU-based prototype:

- The part of the prototype designated to compute the hot spot will be offloaded to the compute device.

  The hot spot is replaced with all the necessary supporting library calls etc. to connect the host code to the co-processor.

- Secondly, a change to data structures to support custom hardware is required. This change is discussed further in *Section 6.3.3*.

One can implement a System on a Chip (SoC) to compute an entire program using custom hardware. However, the scope of such an undertaking is significant and some sub-routines could run faster on a conventional computer than on the FPGA. Instead, the computational bottle-neck is implemented on the compute device. For this purpose, the fitness function has been selected for acceleration. The decision is based on both empirical evidence from early testing and on analytical evidence which suggests that the complexity of the operation is significantly higher than that of others.

The software model of the FGPA hybrid implementation is illustrated in *Figure 6.7*. Note that it is very similar to the CPU-based prototype with the exception of the fitness evaluation stage which includes an assembly snippet, to call the custom instruction, and dependencies on the Convey Computer third party libraries. The two changes are highlighted in red and green respectively.

Custom hardware evaluates fitnesses over multiple clock cycles. To maintain robust operation a decision was taken to use a mutex in order to serialise access of the co-processor. This introduces minimal overhead because the mutex must only be acquired once per fitness evaluation for the entire population.

Every time a solver needs its population fitness to be evaluated it copies its population to the co-processor and provides a pointer to the hosts location for fitnesses corresponding to this solver. The co-processor processes the solutions and discards them afterwards. Finally, fitnesses are communicated back to the host computer as they are computed by the co-processor.

In order to interleave Convey specific instructions with x86 instructions, a change to the compilation framework is required. The Convey computer provides a custom packaged version of the GCC for this purpose. However, the compiler version is 4.5 – which is significantly out of date with respect to the GCC used for the CPU and GPU implementations. This change requires that the use of lambda abstraction, type inference and standard threading libraries be replaced with older equivalents.

Functions which made use of lambda abstraction are re-implemented using simpler features; types were made to be explicit throughout the entire program; and the pthread library was used explicitly to replace the standard C++ threading library.

6.3.2 Implementation of the DePermute Algorithm

The Convey hybrid computer provides three distinct advantages over a conventional computer:

- the memory bandwidth is much higher than that of a conventional computer and it is optimised for random access;
- The development of custom hardware can reduce overheads in computation and allow for the combination of sequential operations in deep functional pipelines.
- it provides a massively parallel driven computation by allowing for functional pipelines to be replicated many times over;

System Interface and Overview

In undertaking a Convey-based design the developer is required to supply supporting hardware to connect the personality specific logic to Convey provided logic. Convey provided logic is responsible for, and abstracts the details of communicating with the host system, distributing instructions to the Application Engines, a few simple scalar instructions, and making memory reads and writes to co-processor memory.

The functionality of the system is divided into four major blocks, illustrated on the right side of *Figure 6.8*. The blocks on the left side of the diagram are responsible for interfacing with Convey provided logic. These are:

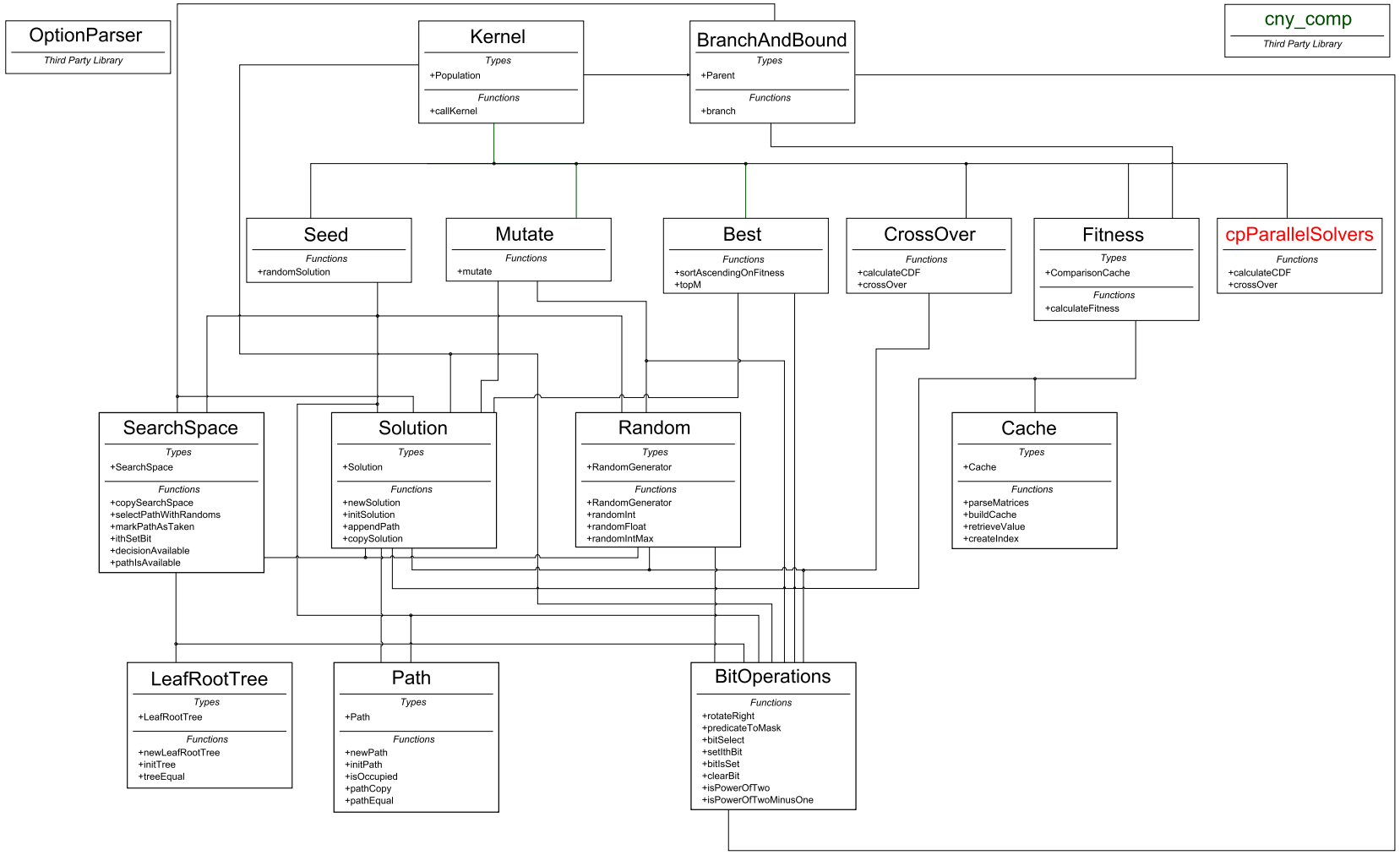


Figure 6.7: Modular decomposition of the FGPA implementation of the DePermute algorithm

- an instruction decoder;
- a register bank;
- an adaptor to abstract over the interface to the logic responsible for evaluating fitness values;

The behaviour of the instruction decoder is trivial and is modelled according to the guidelines outlined in the Convey reference manual [10].

The Convey Reference manual allows for the definition of up to 64 general purpose registers which can be addressed and written to using provided extensions to the x86 language. In the design of the hybrid computer prototype a register bank block is designed as an abstraction over reading and writing to the registers. The interface to the register bank includes:

- a clock signal;
- a reset signal;
- read and write ports;
- a write data port;
- a port which points to all the registers;
- external ports for communicating return data which assembly instructions expect;

The Functional Unit Adaptor is responsible for extracting the signals which Functional Units expect from the general purpose registers. In addition it monitors the state of the controller in functional units to determine whether they have finished computing fitness values.

The state of the computation (i.e. whether functional units are currently computing fitnesses or not) is communicated to the Convey supporting logic via the "stall" and "idle" signals. Where stall halts the host program (for synchronous behaviour) and idle indicates that the custom logic is not currently performing any computation.

Functional Unit Design

The Finite State Machine (FSM) abstraction is used to abstract over the details of operation for evaluating fitnesses. This is a powerful abstraction because it describes, for every possible state that the machine can hold, an appropriate state change and external actions. The functionality of fitness evaluation is decomposed into three modules:

- The controller; which is responsible for implementing the four level nested loop of fitness evaluation.
- The memory controller; which is responsible for the reading of cached values, solutions, and for the writing of fitness values.
- The "index pipeline;" which is responsible for computing the indices of cached distance values.

The last two modules make use of the FSM abstraction in their design.

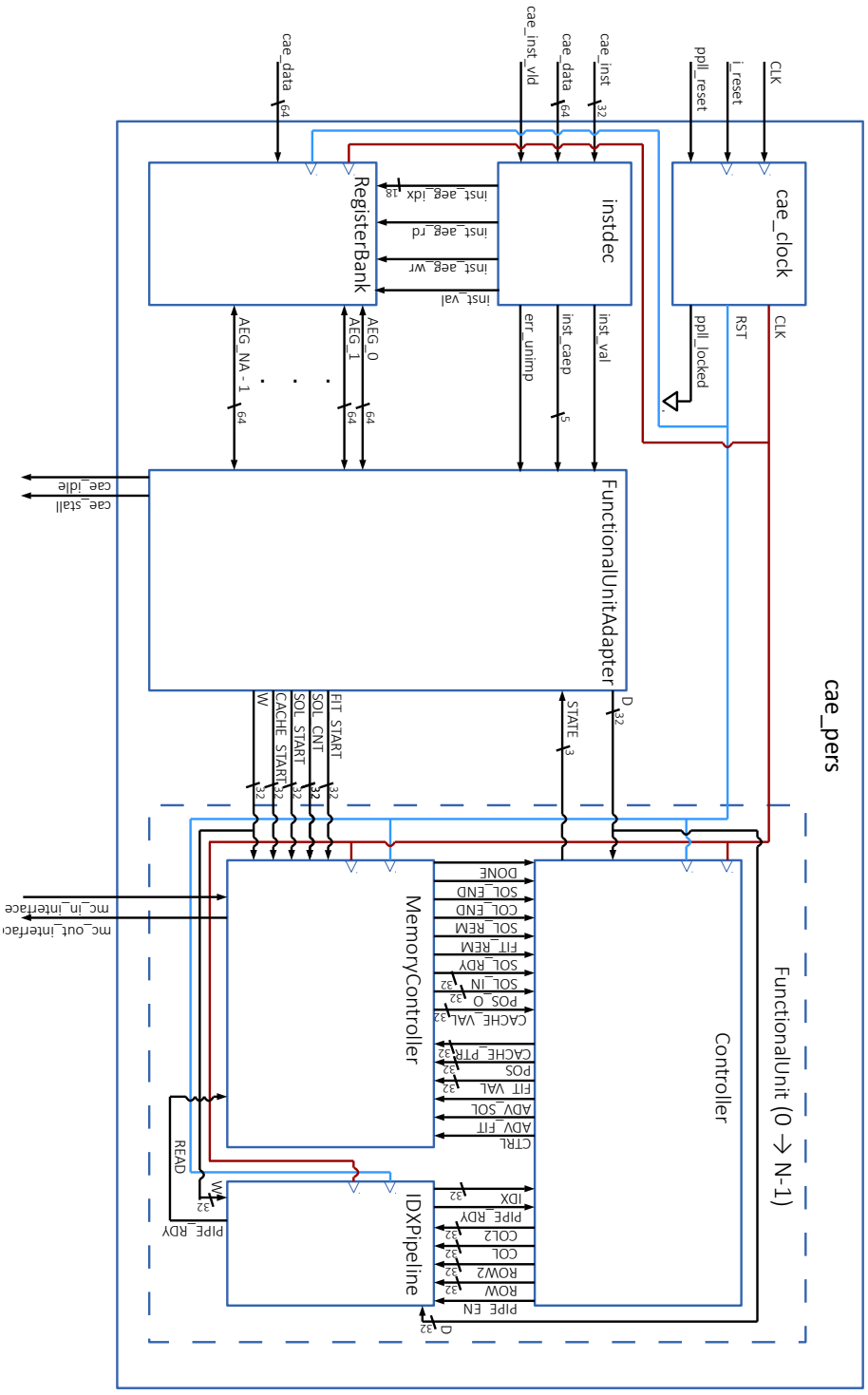


Figure 6.8: Block diagram for the custom hardware implementation of the DePermute algorithm

Controller

The operation of the controller block is described by the FSM in *Figure 6.9*.

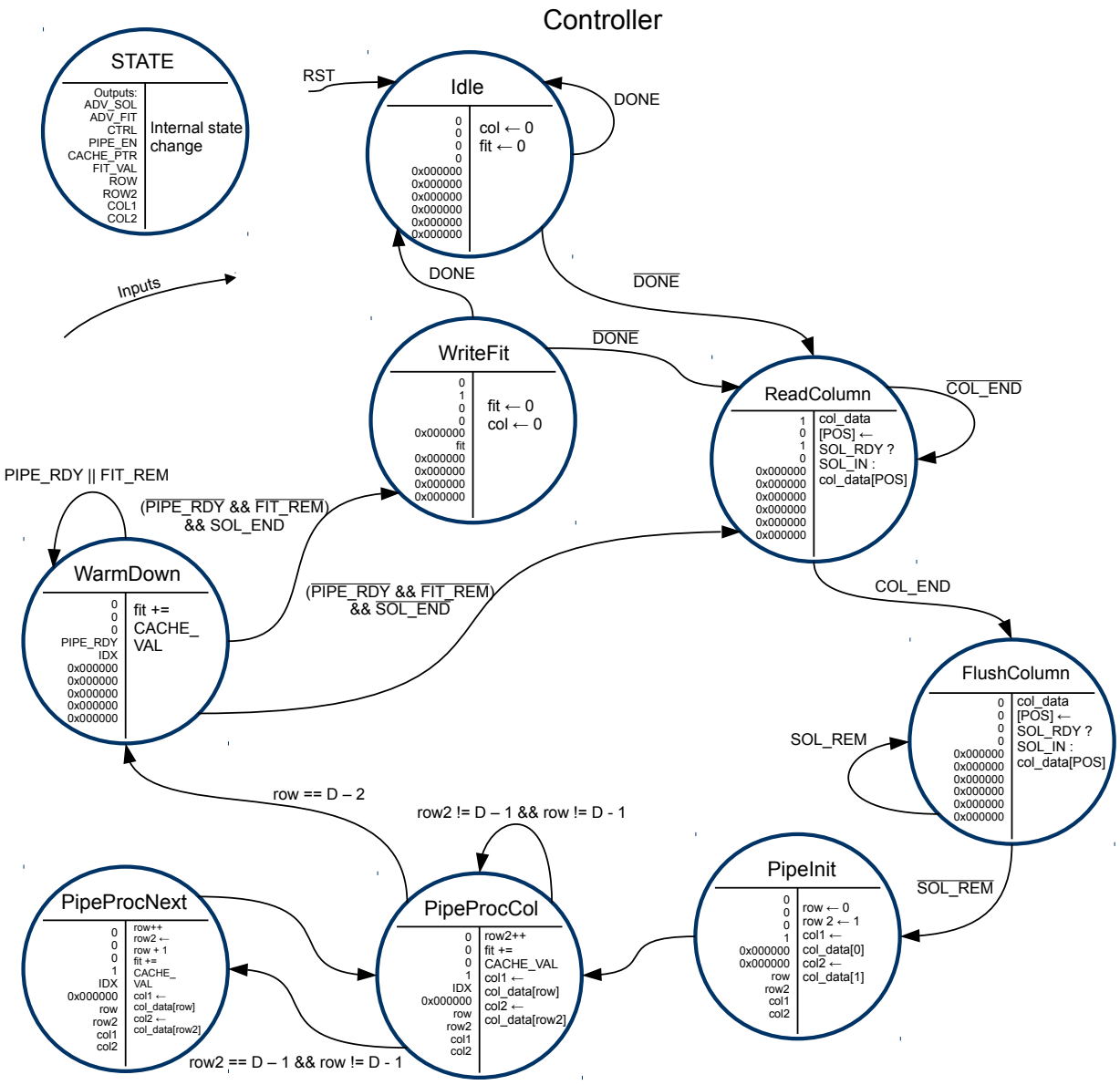


Figure 6.9: State diagram for the Controller FSM

Note that non-standard means were used to illustrate the behaviour of the FSM for the controller (and that in subsequent cases similar nomenclature is used to describe FSMs).

The semantics of state bubbles and transitions remain as they usually are. However, and as the legend indicates, there are three parts to each state bubble. The first two are conventional, being the name of a state and its "Moore type" outputs (i.e. outputs which occur when states are arrived at).

The third part of the state bubble is the internal state change associated with that state.

This part of the bubble models changes to variables. This feature isn't necessary to describe the behaviour of the system, because a variable can be modelled by a different state for each value it could take on. However, the use of states to represent values for variables would lead to an exponential explosion in the number of states in the FSM.

A reset occurs at the start of operation (when the custom instruction is called) sending the machine into the Idle state. Once the machine receives the $\overline{DONE}$ signal it moves into the ReadColumn state. During the ReadColumn state, a column of a solution matrix is loaded from the co-processor device memory into the FGPA chip for fast access.

This is achieved by asserting the memory controller CTRL and ADV_SOL signals, thus selecting solution reading as the memory controller behaviour and advancing the solution pointer respectively. The controller continues reading column values until it is signalled that the column has ended by the memory controller through the COL_END signal. Subsequently the controller blocks while waiting for all column values requested to arrive back from the memory controller in the FlushColumn state.

A simple state is then entered which starts the cache index generating pipeline. The PipeInit state prevents the machine from loading invalid pipe read values into the pipeline.

After a single cycle, a two state loop is entered. One state, PipeProcCol, loads values corresponding to column comparisons into the index generating pipeline. It does so using two loop variables, row and $row2$, where $row2$ corresponds to the inner-most loop in the fitness evaluation routine and row corresponds to the second inner-most loop. row and $row2$ are used to index the loaded column to retrieve corresponding symbols in the solution matrix. row, $row2$ and their corresponding symbols along with the constants W and K are sufficient to generate a cache index.

Simultaneously the controller adds the current cache value loaded from the memory controller to a running total. This value is accumulated regardless of whether or not one has arrived back, because there is no performance penalty in doing so. (Addition is commutative and the memory controller returns zero when a value has not yet arrived back.)

When the value of $row2$ reaches $D-1$ the inner-most loop terminates (conceptually) and the state changes to PipeProcNext. PipeProcNext advances the row variable and sets $row2$ to $row+1$ – as the second most inner loop would have done in the software model. It also makes the next cache value request to the index generating pipeline and adds the retrieved cache value to the running total.

Once the value of row reaches $D-2$ the outer most loop terminates (conceptually) and the state of the machine changes to WarmDown. During WarmDown, the machine waits for all cache values to arrive back from memory. The controller is made aware of whether there are fitness values not yet returned by the FIT_REM signal from the memory controller.

At the point when the fitnesses have all returned, the machine enters one of two states. If the memory controller flags that an entire solution has been read then the machine moves to the WriteFit state, where it requests for the current fitness to be written and resets the fitness value. Otherwise it continues to process the next column from the ReadColumn state.

If all solutions have been read, as flagged by the DONE signal, then the machine returns to the Idle state.

Memory Controller

The Memory Controller consists of four sub blocks. Namely:

- the Memory Adaptor;
- the Solution Loader;
- the Cache Loader;
- the Fitness Writer;

Whilst only one block can drive a signal to the Convey memory controller, three sub-blocks inside the Memory Controller block require access to the Convey memory controller. The Memory Adaptor is responsible for providing a switched interface to the Convey memory controller to each of the other three sub blocks. This is accomplished using a multiplexer. The order of precedence for each type of operation (which the memory controller serves) is as follows:

- If ADV_FIT is asserted, then the write interface is switched to the Fitness Writer.
- If the CTRL signal is asserted and ADV_FIT is not, then the Solution Loader is selected.
- Finally, in the case of neither signal being asserted, the Cache Loader is selected.

The arrangement of the four blocks in the MemoryController is illustrated in *Figure 6.10*.

The operation of reading and writing to the Convey memory controller is abstracted by two FSMs – the Abstract Reader and the Abstract Writer FSM. The FSMs provide asynchronous reading and writing and a simplified interface for the SolutionLoader, CacheLoader and FitnessWriter to use. The two FSMs are illustrated in *Figures 6.11 & 6.12*. Note that the FSMs are described using the same nomenclature as the FSM for the Controller.

The operation of these FSMs is simple:

- Remain idle until a request is made.
- When a request is made, make the write/read by moving to the Write/Read state only if the Convey memory controller can process another request (flagged by mc_rq_stall).
- Otherwise move to halt and stay in the halt state while the Convey memory controller is busy.

The AbstractReader FSM also contains a queue of identifiers for requests being made so that the Controller can interpret the results out of order.

In addition to these FSMs the abstract writer and reader contain a queue which stores the requests made by the Controller. This allows for the machine to occupy the Halt state without missing any requests. When a write is made, the machine "pops" the value off of the top of the queue and continues attempting to make writes until the queue is empty – as flagged by EMPTY.

The Solution Loader extends the functionality of the abstract reader by adding two counters. A counter for which *row* is being read in a column and a counter for the column being processed in the solution. The counters allow for the SolutionLoader to flag column and solution end states to the Controller. The arrangement of counters and FSM blocks for the Solution Loader is illustrated in *Figure 6.13*.

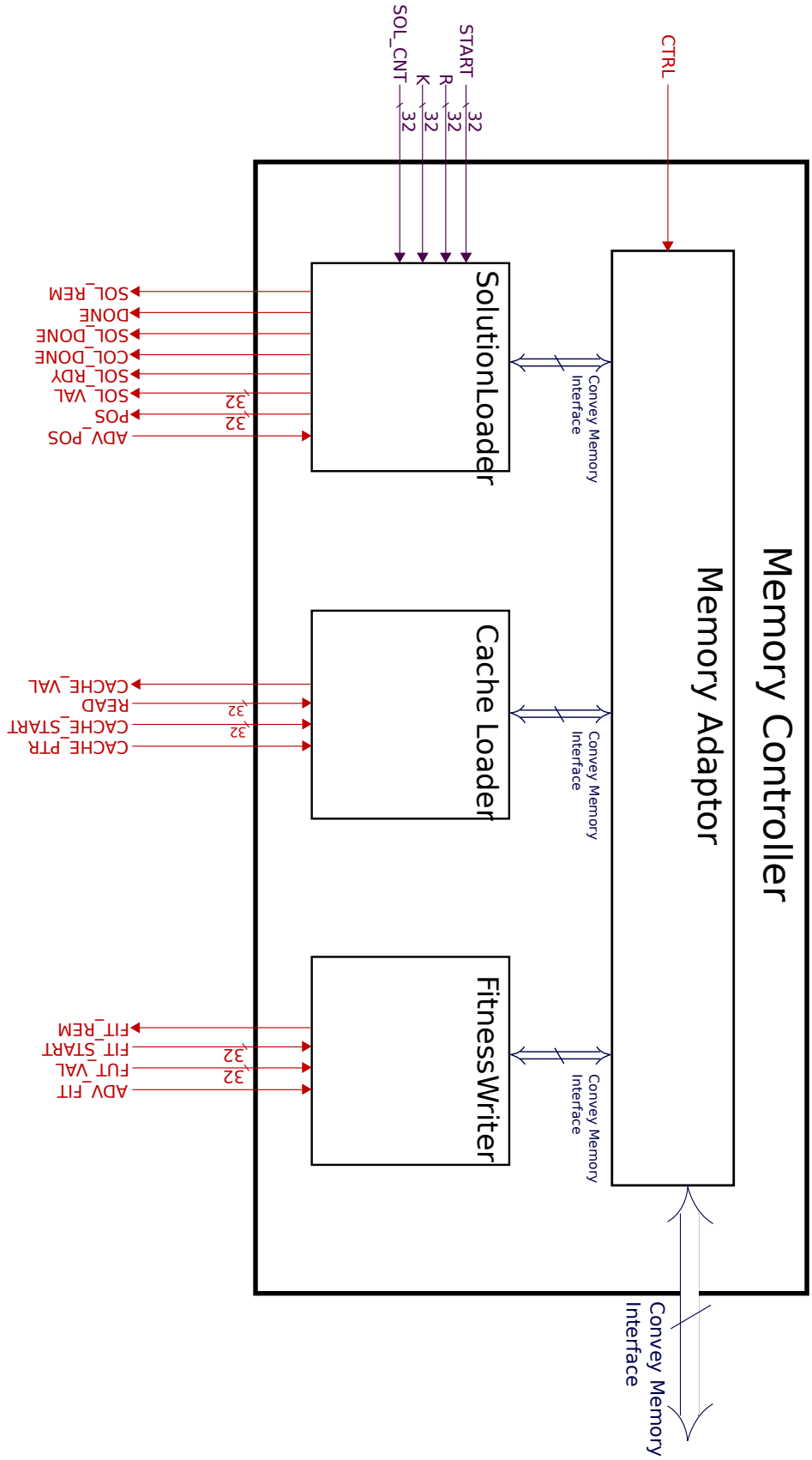


Figure 6.10: Block diagram of the memory controller sub blocks

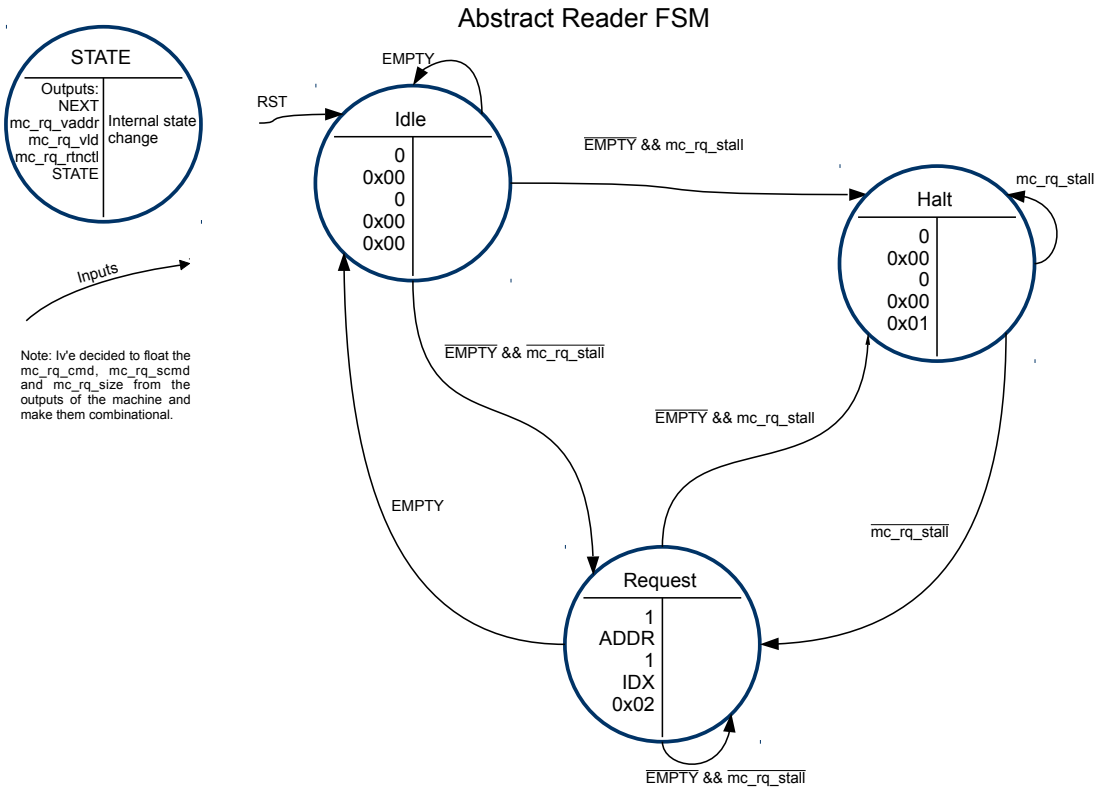


Figure 6.11: State diagram for the Abstract Reader FSM

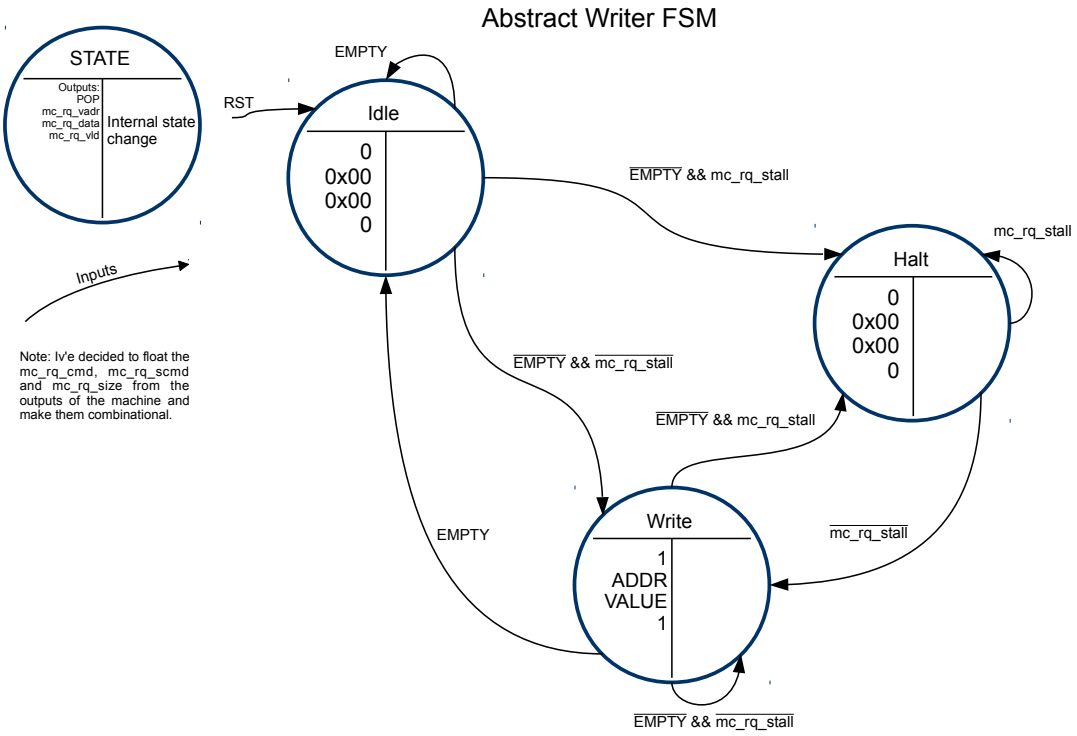


Figure 6.12: State diagram for the Abstract Writer FSM

The Cache Loader extends the functionality of the abstract reader by ensuring that it always returns a zero value when there is no return from the Convey memory controller. The arrangement of the Cache Loader FSM and conditioning block are illustrated in *Figure 6.14*.

Finally the Fitness Writer extends the functionality of the Abstract Writer by keeping track of the current position of the pointer into fitness values and incrementing it whenever a request is made. The arrangement of the Fitness Writer FSM and supporting logic is illustrated in *Figure 6.15*.

Index Pipeline

The computation of indices in the cache involves 7 multiplications, 3 subtractions, 4 additions, 1 division and 1 negation – all of which are integer operations.

The propagation delay which this arithmetic incurs has been found, by experimentation, to be too large to fit in a single cycle of the FPGA. To counteract this effect (without incurring large performance penalties) the steps involved in the arithmetic were analysed for data dependencies. Groups of operations which have data dependencies are arranged into five stages in a pipeline. This reduces the number of cycles needed to compute an index to 5 from 16.

In addition, registers can be added between each of the stages in the computation so that as stage 0 finishes for the first request the next request can begin stage 0 and so forth for the other stages in the pipeline. In doing so the pipeline can handle up to five requests simultaneously. After a warm up period of five clock cycles, a request is finished after each

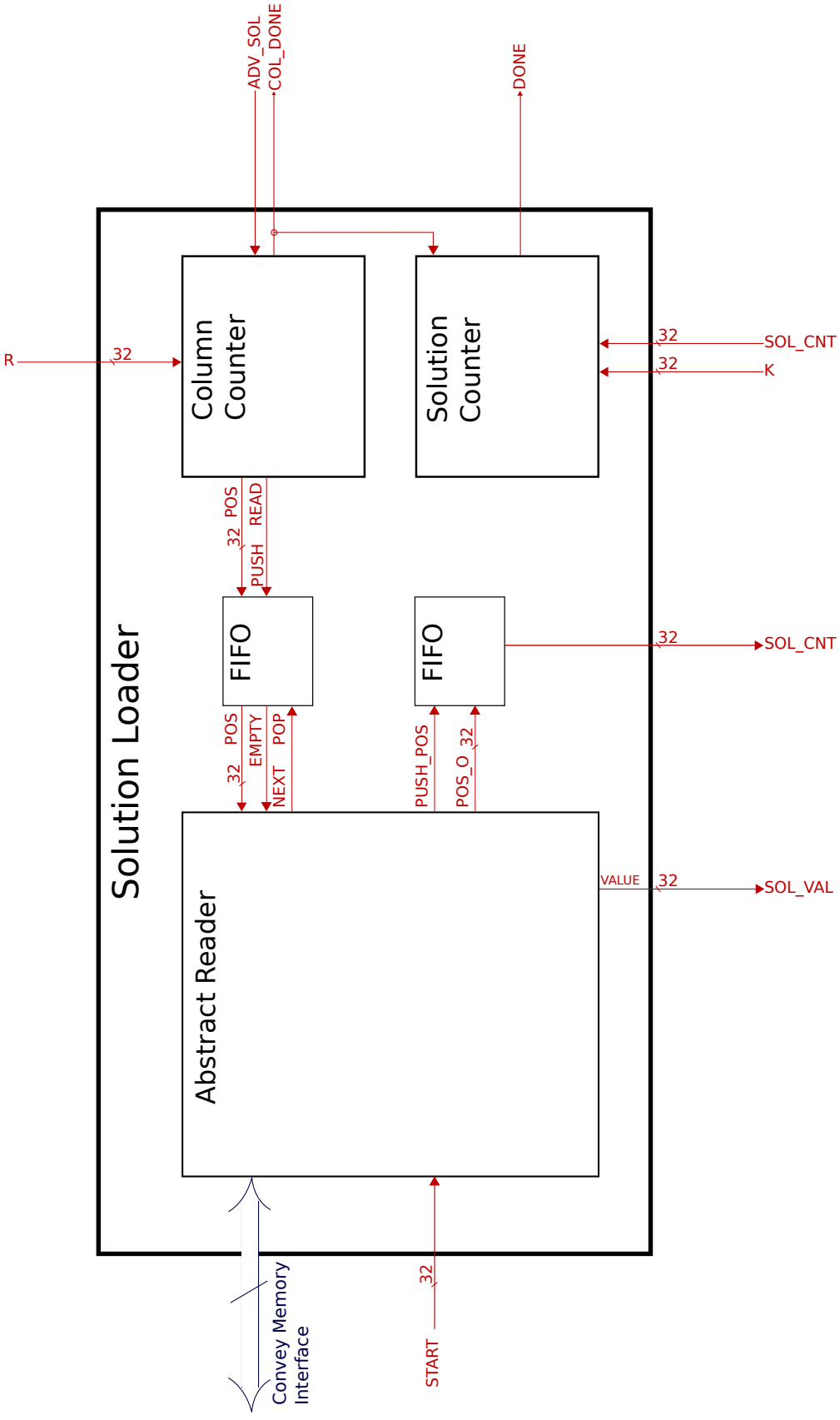


Figure 6.13: Arrangement of FSM blocks in the Solution Loader block

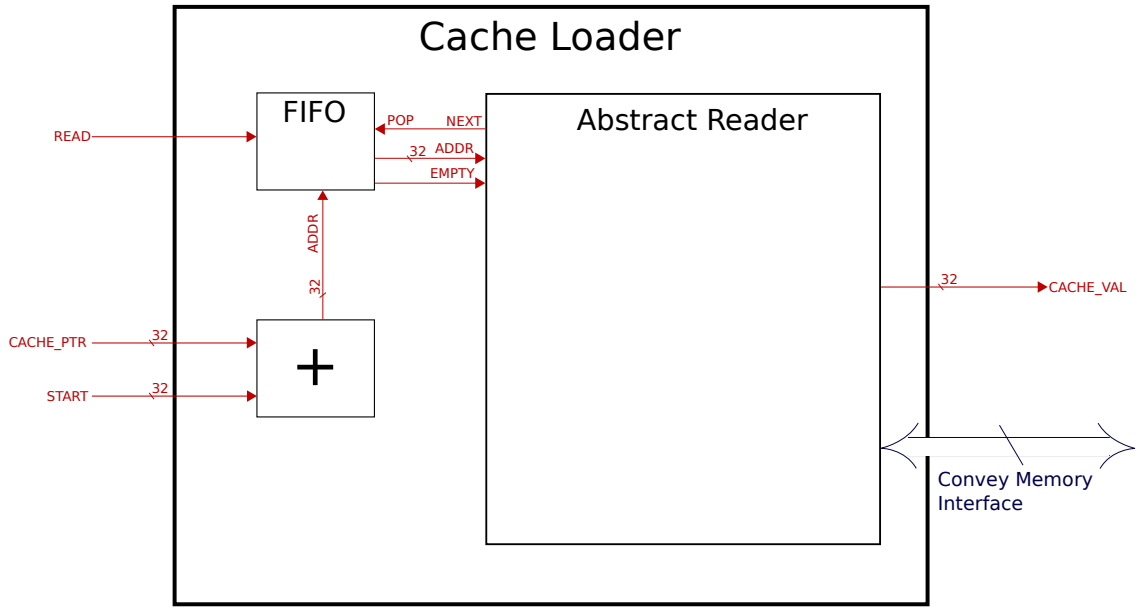


Figure 6.14: Arrangement of FSM blocks in the Cache Loader block

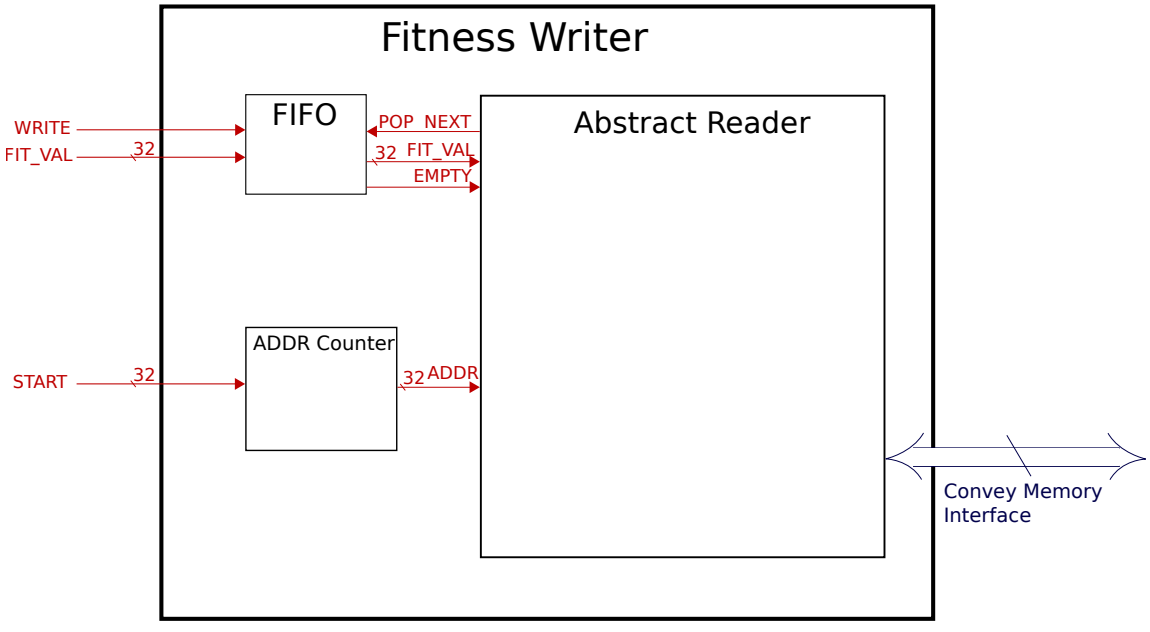


Figure 6.15: Arrangement of FSM blocks in the Fitness Writer block

clock cycle until all requests have been processed. Overall $16n$ is reduced to $5+n$ clock cycles to process every element.

6.3.3 Data Structures

It is simpler and more efficient for a Functional Unit's Memory Controller to read values from device memory in column major format. However we would prefer for there to be minimal changes made to the CPU-based prototype in order to reduce required work in porting between the two formats.

Therefore the Solution data structure is left as it is for the CPU version, and the Solution is *constructed* in a different way. Solutions and populations are allocated as one continuous block, as they are in the GPU prototype. Subsequently, the FGPA host program constructs solutions in its format by extracting pointers to the starts of columns, solutions and populations and using them in the same format as was used in the CPU prototype.

The combination of the two data structure implementations is less efficient in space, but requires the least changes to the existing code base while supporting fast traversal on the hybrid computer.

6.4 Test Objectives

In [21] research was conducted into experimental practices in computer science. The research investigates whether a number of published scientific experiments involving computer based testing are in fact robust. It was found that there are many instances of inadequate detail in testing procedures in the literature and that in some instances this had led to claims which were tenuous at best and in many instances were incorrect [21].

The stated goal of this testing procedure is to provide a fair comparison between all programs. Considerable care has been taken in the formulation of the testing procedures in order to avoid the pitfalls of flawed testing procedures found in the literature. The protocol followed is as follows:

- The details of the model of computer used are stated.
- A description of the build tools (including compiler, flags, and operating system) are provided.
- Data sets used in the testing procedure are provided.
- A discussion about solution quality in circumstances where heuristics may produce different solutions is provided.

Finally it is suggested that multiple machines are used in the testing procedure so as to illustrate the difference in computational characteristics across platforms [21].

6.5 Time Based Testing Procedure

The discussion highlights the particular programs being tested, how the programs were compiled for the testing procedure, data sets which the programs were run on and the exact procedure for executing the program.

A script was written for all testing procedures which automates each procedure. The scripts are written in the Bash programming language and use programs written in Python and Haskell for various tasks. These scripts are available in *Appendix C*. For the Windows operating system, the git bash executable was used to execute bash scripts. Note that the versions of all programs used is listed in *Appendix D*.

6.5.1 Programs Under Test

The following three programs are selected for time based comparative testing:

1. the CLUMPP program;
2. the CPU based prototype for the DePermute algorithm;
3. the GPU based prototype for the DePermute algorithm;

The CLUMPP program is packaged with three algorithms. A full enumeration algorithm, which we have elected to not test, a greedy algorithm termed Greedy and a very greedy algorithm termed Large K greedy. In the results section Greedy is referred to as Accurate CLUMPP and Large K Greedy is referred to as Fast CLUMPP. *Table 6.1* lists the parameters used for Accurate and Fast CLUMPP respectively (the remaining parameters are problem specific or irrelevant for testing purposes):

Parameter	Value
Accurate CLUMPP	
Weight (W)	1
Similarity Statistic (S)	2
Datatype	1
Repeats	1000
Method (M)	2
Fast CLUMPP	
Weight (W)	1
Similarity Statistic (S)	2
Datatype	1
Repeats	1000
Method (M)	1

Table 6.1: Table of parameters used for Accurate and Fast CLUMPP

Two parameters have the greatest impact on the performance of CLUMPP. These are:

- selection of algorithm;
- repeats;

The repeats parameter controls the number of runs which the program makes, from random starting positions. This parameter has a direct impact on CLUMPP's performance. One would expect the running time of the program to be linearly proportional to the repeats parameter. It also impacts the quality of solution found.

In their paper, Jakobsson et al.describe how the quality of solution is greatly impacted by the number of repeats [24]. It was found that by varying the parameter between 100 and 30000 a difference in quality of approximately 0.01 was observed. This difference is significant because it could equate to several misalignments in solutions.

This dissertation is also interested in the running time of the program and a selection of appropriate quality must be balanced by performance in order to draw a fair comparison. One thousand repeats was chosen for this experiment because it ought to yield acceptable quality in solution while still being an order of magnitude faster than the upper bound on repeats which was tested by Jakobsson et al.. It's important to note that a lower number of repeats might result in a more competitive result, but that this would come at the cost of quality in solution.

6.5.2 Test Platforms

A sample of different computer platforms was used to run tests. In this way it was hoped that biases which a computer might have toward the over represented computations in either program were mitigated to some extent.

Overviews of the test platforms are listed below:[2]

- System A:
  - Intel Xeon E2-2670;
  - 2.6GHz-3.3GHz;
  - 20MB Cache;
  - 8 vCPU cores;
  - 15GB RAM;
  - NVIDIA Grid K520 Graphics Card;
- System B:
  - AMD FX9370;
  - 4.4GHz;
  - 8 Phyiscal Cores;
  - 8MB L3 Cache;
  - 16GB RAM;
  - NVIDIA Geforce GTX960;

[2]Refer to *Appendix E* for greater detail.

- System C:
  - E5-2620;
  - 2.1GHz;
  - 15MB Cache;
  - 32 GB RAM;

6.5.3 Program Compilation

Each program is recompiled with appropriate settings for the platform it is run on. The following compilers were used to compile binaries:

- MS Visual C++ 2013 Community Edition v12.0.31101.00;
- NVIDIA CUDA Toolkit v7.1 & v6.9;
- GCC Version 4.8;
- OpenCL Version 2.0;

Compilation for the CPU based prototype uses the following flags in GCC.

- `-O2`

Compilation for the CPU Based prototype uses the following flags in MS VC++.

- `/Ox`
- `/Ob2`
- `/Oi`
- `/Ot`
- `/Oy`
- `/GT`
- `/GL`

No additional flags are supplied to the OpenCL compiler because it has safe optimisations enabled by default.

6.5.4 Data Sets

Two categories of data set are selected for testing:

- randomly generated data;

  Three problem sizes are considered in testing procedures. In all instances there are five hundred individuals involved in a hypothetical population structure study. R is set to 20, as is common practise, and K is set to even intervals from 10 to 20. Finally, to simulate varying results in Q-matrices, a noise level of $\frac{0.01}{K}$ is introduced (refer to *Section 6.6* for a complete description of the implementation of noise in data sets.)

- real data set;

  The real data set aggregated from various population structure studies. More detail can be found in *Section 6.7*

  - The sample contains 500 individuals .
  - The ADMIXTURE program was run 50 times for each value of K from 2 to eleven.

6.5.5 Methods of Testing

The testing procedure is automated by a series of BASH scripts which are listed in *Appendix C*. For random data based testing the procedure was as follows. Given a file describing the parameters, K, R, M etc. for the desired problems to test:

1. Generate a series of randomised data sets as described by the procedure given in *Section 6.6*.
2. For each value of K and for a predefined number of repeats, run the program under test and log the time taken using the `time` command.

A similar process is used for real data set based testing with a difference to step number one:

1. Copy test data from a central location to a scratch space and format it into the required format.
2. *remains the same as in the randomly generated data testing procedure;*

The experiments on real data sets were repeated twice and the experiments on random data sets were repeated three times. In each case the average of the running times and quality was recorded.

6.6 Random Data Set Generation

A program was written to generate simulated random Q-matrices. The program generates random Q matrices as inputs for the experiment. In *Chapter 5* a methodology for the generation of random Q-matrices was established. The method is used again in this chapter. Another overview of the process is presented for convenience:

1. Generate a random Q-matrix of an appropriate size, and then:

   Repeat the following M times (fixed to 100 for the following experiments experiment):

   (a) Sample K random numbers in $(0, 1)$ under a uniform distribution (i.i.d.).

   (b) Normalise the random numbers.

   (c) Append the random numbers to the prototype matrix.

2. Copy the prototype matrix R times.

3. For each value in the matrix add a random real number x, where $x \in (0, n)$.

4. Normalise the rows of each matrix again.

5. Perform a Knuth Shuffle on the columns of each matrix.

Chapter 5 intended to establish computational characteristics of DePermute in terms of the heuristics used and the number of iterations taken by the algorithm. This chapter adopts a more practical approach by establishing the running time of DePermute and CLUMPP run on randomly generated data and data sampled from test subjects.

The method of random problem generation does not simulate population structure. Randomly generated data sets are intended as a fair measure of how DePermute and CLUMPP perform given data which doesn't show structure. Data without obvious structure presents a challenge to algorithms and allows us to establish the performance of algorithms under worst case conditions. Tests using randomly generated data and data sampled from test subjects allows for comparisons to be drawn between the performance of the algorithms when confronted with difficult data and data – which one is more likely to encounter in practise.

6.7 Practical Data Set Acquisition

A data set was assembled from various projects by Scott Hazelhurst. The data set includes samples from the Thousand Genomes Project, Human Genome Diversity Project, the Singapore Sequencing Malay Project, Southern African Human Genome Project and a research project into the diversity of Black Southern Africans [7, 43, 53, 8]. The data includes the values of SNPs and phenotypical information sampled from 500 individuals. The subjects of the data set originate from a total of 40 suspected population groups and one of the individuals has an unknown population group. The population groups from which the samples were acquired are listed in *Appendix B*.

The ADMIXTURE program was run 50 times on the data set for each value of K from 2 to 11. The results were recorded in files stored in separate directories. The data in each directory is aggregated into a single file by concatenating the files for each corresponding problem size. The data is then broken down into a series of matrix files for processing by DePermute and also transformed into a format suitable for CLUMPP.

Test data was persisted to a revision control system along with the scripts used for running CLUMPP and DePermute so that the testing environment could be replicated exactly across all testing platforms. The process also accommodated for minor changes in testing scripts to be redeployed faster.

Chapter 7

Results

This chapter presents all results generated during the testing procedure described in *Chapter 6*. The results are organised into sections corresponding to the machine on which the tests were conducted (i.e. Systems A, B and C) and then into sub-sections according to the type of data used in the experiment, that is, each section contains a sub-section for the results achieved when running the program for randomly generated data and then for the real data set. Also note that the quality results of systems B and C appear in *Appendix A* because quality is much less a function of the platform running the experiment.

Detailed results are preceded by a summary section which includes two summary graphics. The summary is based on the results achieved on system A because that result set is the most complete of the three result sets. Further remarks are given at the start of the section on the results generated by system A and differences are noted at the start of the sections on results for systems C and B.

7.1 Summary of Results

The average running time and quality of result for the experiments run on real data is summarised in figures *Figures 7.1 & 7.2* respectively.

The *first figure* indicates the relative time taken by each program with respect to the longest running time – averaged across all problem sizes. Note that the running times indicated do not reflect actual experimental results. On the figure, experimental results are aggregated and scaled to fit on the 12 hours of a clock face – for illustrative purposes. i.e. the ratio of running times between CLUMPP and CPU-DePermute, and then between CLUMPP and GPU-DePermute was computed for each problem size and averaged respectively. Finally the averaged ratios were scaled by a factor of twelve so that the longest running program appears as twelve hours and the other programs under test each appear as times within twelve hours.

The graphic indicates that, for experiments presented, if the CLUMPP program is run using the Large K-Greedy algorithm and its average running time is scaled to equal twelve hours then the CPU and GPU versions of the DePermute will have running times of 37.47 minutes and 1.25 minutes respectively.

The *second figure* indicates the relative quality which each program achieves. The centre of the target represents the solution with zero average distance between all matrices (i.e. the

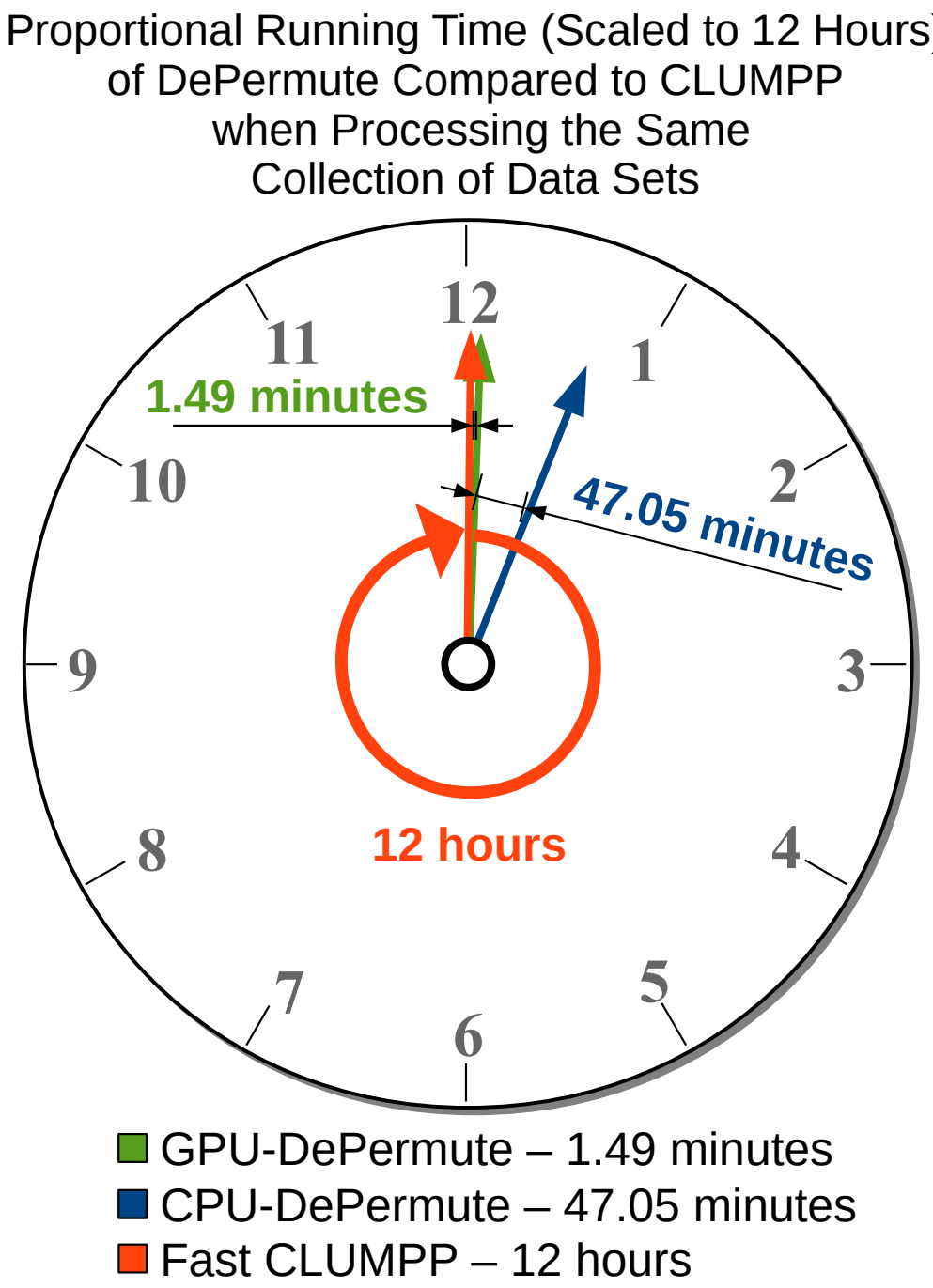


Figure 7.1: Graphical summary of the time based testing results achieved by CLUMPP and DePermute

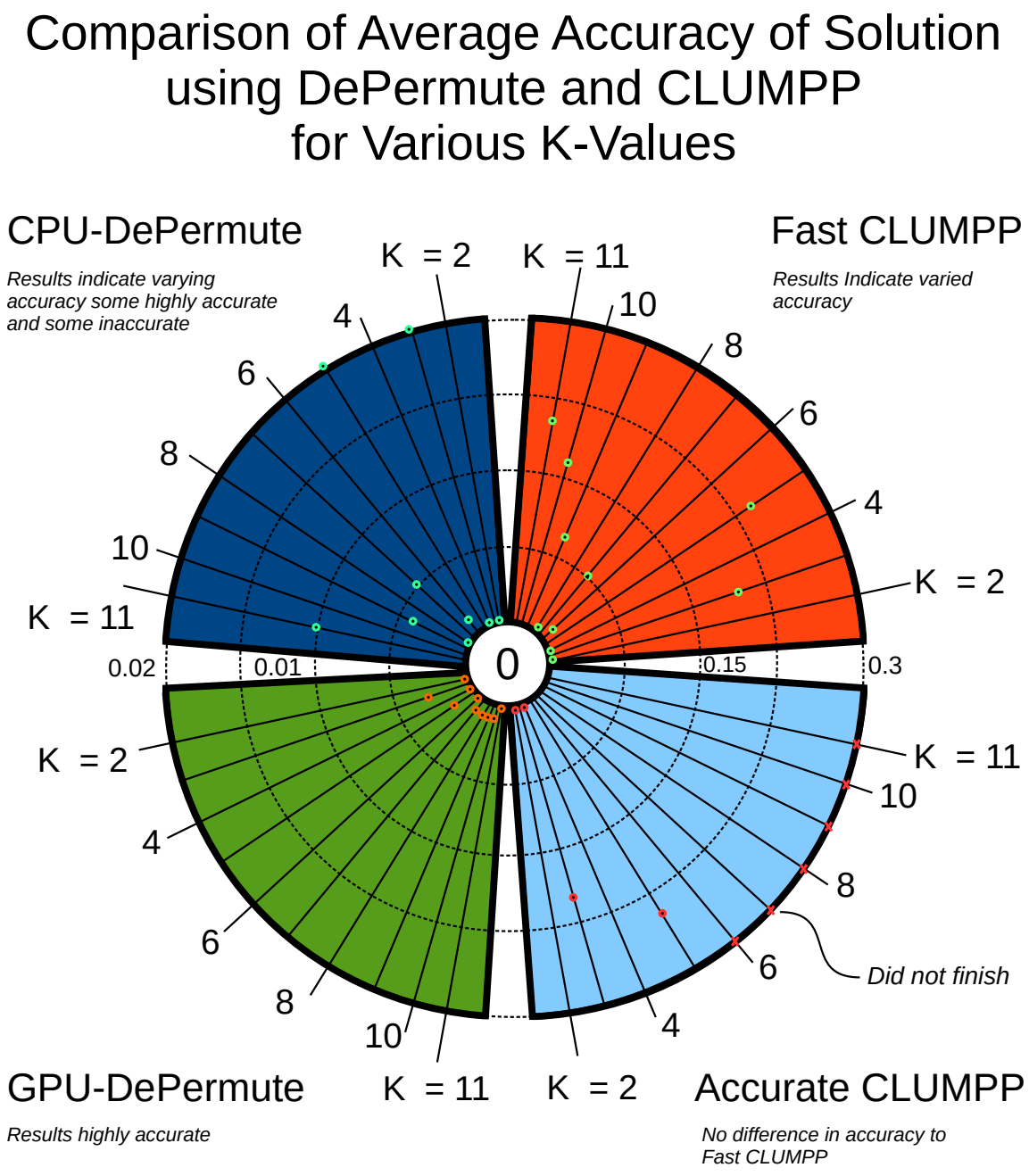


Figure 7.2: Graphical summary of the quality of results achieved by CLUMPP and DePermute

"perfect" solution). It is important to note that:

- There are differences in the measure of the quality of a solution between CLUMPP and DePermute.
  - The function used as a heuristic to measure quality is marginally different.
  - DePermute denotes a quality of zero as perfect.
  - CLUMPP denotes a quality of one as perfect.
  - In the graphic the CLUMPP quality has been altered such that a quality of zero is denoted as perfect:
    * $QD = 1 - QC$
      Where:
    * QD is the Quality as indicated on the diagram.
    * QC is the original Quality printed to the user when using the CLUMPP program.
- The perfect solution may not exist for a given data set.

A Note on the Data

The data set for the $K = 11$ problem size was not complete. Only 41 runs of ADMIXTURE were present. It is worth keeping this in mind when perusing the results, because the $K = 11$ run completes in less time than the $K = 10$ run across the board. The $K = 11$ result cannot be compared (fairly) to the other results.

Observations

Note that CLUMPP finds solutions of a highly consistent quality across multiple runs for the same problem. In addition, note that no difference in quality was recorded between the high accuracy heuristic settings for CLUMPP when compared with the faster heuristic. Finally, note that the opposite is true of DePermute, i.e. DePermute usually finds slightly different solutions across multiple runs of the same problem. This may be caused by one of two effects:

1. There are multiple solutions to the problem with sufficiently high quality which are found on different trials by DePermute.
2. A classic heuristic can easily find the same solution on multiple trials due to the deterministic nature of the search it conducts.

DePermute is faster than CLUMPP in most cases. DePermute has also been shown to be capable of computing the solution to much larger problems in a reasonable time frame without compromising on the quality of solution found.

DePermute uses significantly more memory than CLUMPP, because DePermute actively trades memory efficiency for more opportunities to employ data parallelism. DePermute requires memory proportional to K and R (as a product with P) to support initial populations large enough to converge on a high quality solution fast. In practice, we find that CLUMPP requires memory in the order of megabytes and that DePermute requires memory in the order

of tens of gigabytes in order to process the larger problems in the test data sets used in these experiments.

Both the CLUMPP and DePermute programs can can be tuned to produce lower quality results in a shorter amount of time. Smaller LSP problems ($K \approx 5$) can be run by DePermute on personal computers with equivalent quality to that which was produced in the experimental process outlined by this chapter. Larger problems ought to be run on high power clusters and on graphics cards with sufficient specifications. It's reasonable to assume that the researcher who is interested in larger LSP problems will likely have access to sufficient hardware (and time) to deal with larger problems.

The quality of results generated by DePermute and CLUMPP vary to a very small degree. It is not clear whether the variance in quality of solution generated by DePermute is unfavourable or whether the solutions it generates are closer to the ontologically correct solution.

The real data set contained problems with a relatively low complexity where the solution was often found in a very short time frame – the complexity of the problem was often reduced through block alignments. For instance in the case of $K = 2, 4, 6\&8$ a high quality solution was found entirely by block alignment[1]. This result confirms that the block alignments algorithm is a valuable component of DePermute. It also hints that real data may not be complex in many instances and the real challenge may instead lie in the selection of an appropriate value of K.

7.2 System A

The results produced by system A were highly consistent, i.e. little to no fluctuation was measured between subsequent runs of the programs.

System A comprised a high performance CPU and GPU, sufficient memory and an SSD. This minimised the penalty for reads from persistent memory and gives a fair platform for both CLUMPP and DePermute to be run and compared.

7.2.1 Randomly Generated Data

We note that the performance of CPU-DePermute and the faster configuration for CLUMPP are comparable. CPU-DePermute achieved a lower time for the smallest of the random problem sizes, but performed slower than CLUMPP for all others. Also note that GPU-DePermute completed its tasks in up to two orders of magnitude less time than the faster setting for CLUMPP. Note when viewing *Figure 7.6* that the vertical axis is a logarithmic scale.

The quality of result achieved by CLUMPP is consistent across all problem sizes *(note that a result closer to zero is better for DePermute, but that a result closer to one is better for CLUMPP)*. The quality of result achieved by DePermute varies for each problem and is greater for GPU-DePermute than for CPU-DePermute.

We note that further analysis of the quality of result in a biological context would serve as a better grounds for comparison between the three programs under test. In this context the result achieved by the GPU version of DePermute in contrast to the CPU version is

[1] It is curious that block alignments are found on multiples of two although no explanation for this anomaly is presented.

emphasised. It is hypothesised that result owes itself to a greater degree of exploration happening on the GPU through parallelism than on the CPU (which must sequence some of the searches when there are too few cores to execute searches simultaneously).

Table 7.1: Table of *Time* per group size (K) for randomly generated data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system A

K	CPU-DePermute (seconds)	GPU-DePermute (seconds)	CLUMPP (seconds)
10	62.89	12.30	95.81
15	419.97	2.46	210.42
20	1106.11	3.72	371.14

Table 7.2: Table of *Quality* per group size (K) for randomly generated data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system A

K	CPU-Depermute	GPU-Depermute	CLUMPP
10	$9.98E{-}03$	$2.41E{-}03$	$9.99E{-}01$
15	$1.06E{-}02$	$4.71E{-}04$	$9.99E{-}01$
20	$9.33E{-}03$	$3.66E{-}04$	$9.99E{-}01$

7.2.2 Real Data

The results for testing using real data are dramatically different to those achieved using randomly generated data. Note two important differences to the randomly generated data testing results:

- CLUMPP, using fast settings, was significantly slower than DePermute across the board.
- The results achieved by CPU and GPU DePermute were similar in most instances.

The differences may be accounted for by the fact that the real data set contains many block alignments, which CLUMPP does not account for, and by the fact that the problems are relatively simple in comparison to the tests on randomly generated data. DePermute is more responsive to the complexity of a problem and will, given a simple problem, halt early.

We note that the quality of result achieved by the CLUMPP program is more variable – in contrast to the tests on randomly generated data[2]. The difference is likely because the data on which the population structure study was made is more conducive to being fit to particular numbers of population groups.

[2]Interestingly one can see the same pattern across multiples of two below ten for quality of solution which was seen in block alignments for DePermute

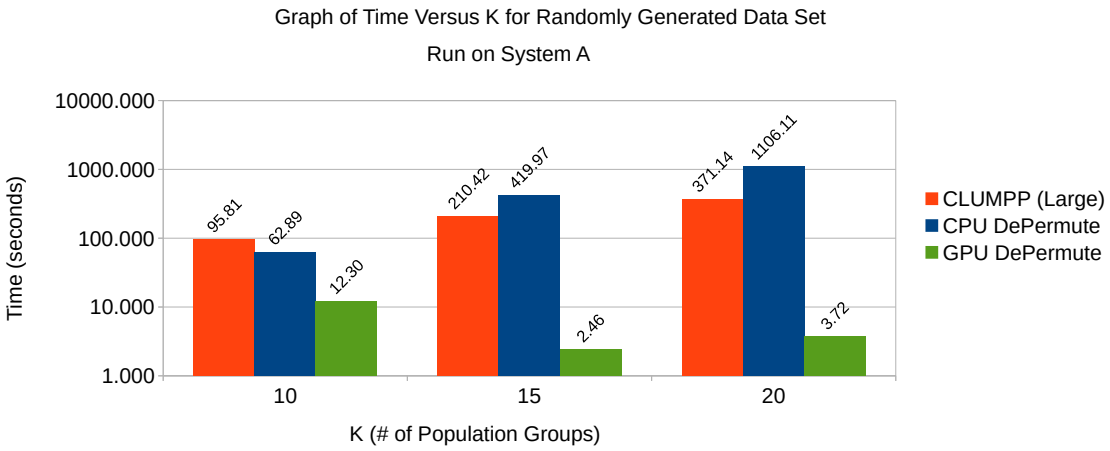


Figure 7.3: Graph of *Time* versus (K) for randomly generated data testing of CLUMPP and DePermute run on system A

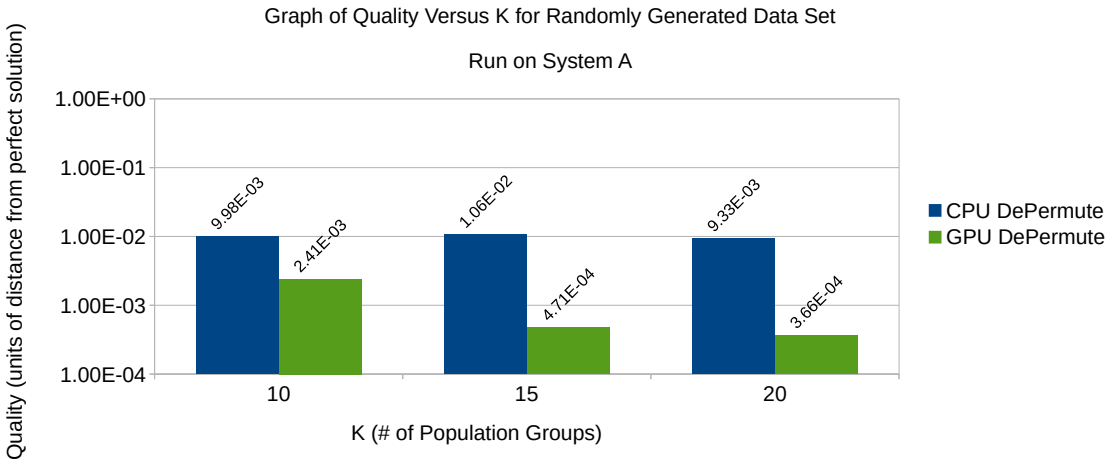


Figure 7.4: Graph of *Quality* versus (K) for randomly generated data testing of CPU and GPU DePermute run on system A

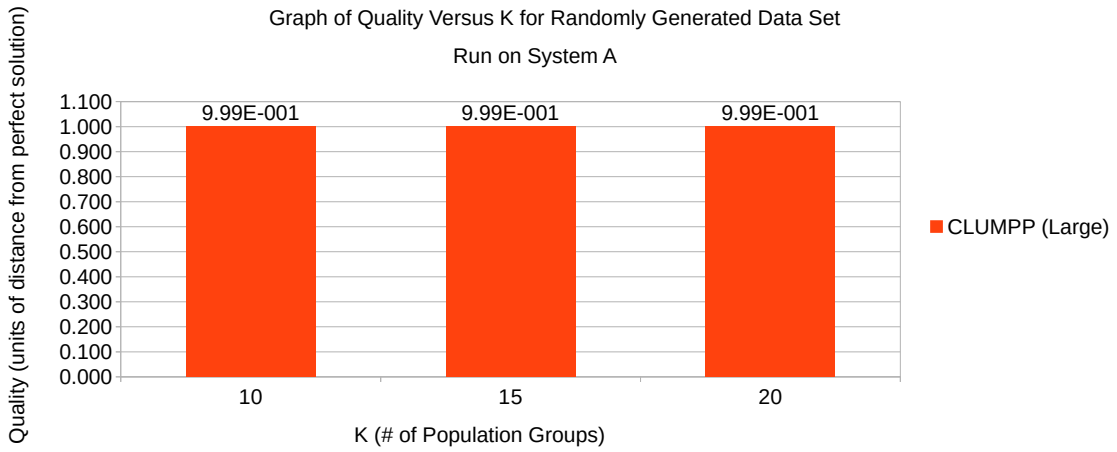


Figure 7.5: Graph of *Quality* versus (K) for randomly generated data testing of CLUMPP run on system A

We note that the results achieved by DePermute are near identical in instances where the problem was solved entirely by block alignment and that in most other instances the results are similar. This may be accounted for by the mechanic of halting once a sufficient quality of solution has been found, that is, the program is more likely to find solutions with quality close to the target stopping quality.

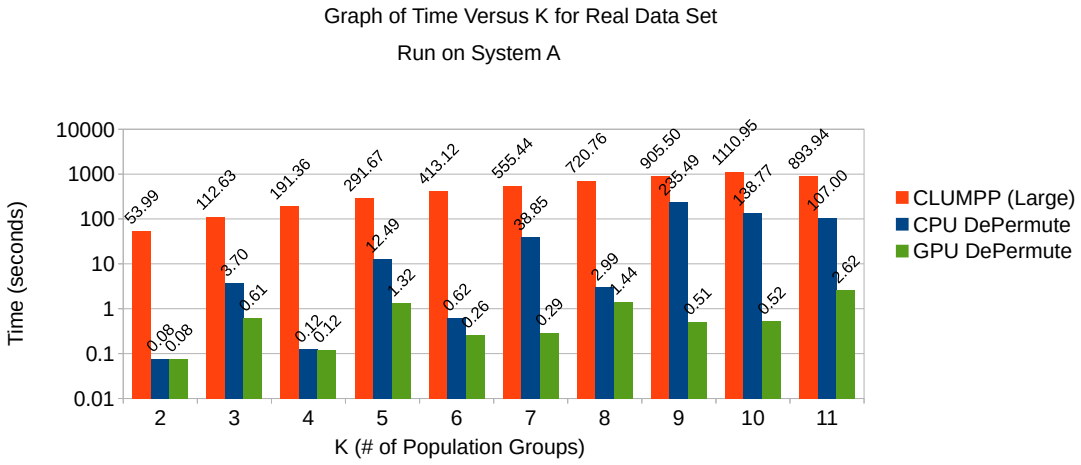


Figure 7.6: Graph of *Time* versus (K) for real data testing of CLUMPP and DePermute run on system A

Table 7.3: Table of *Time* per group size (K) for real data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system A

K	CPU-DePermute (seconds)	GPU-DePermute (seconds)	CLUMPP (seconds)
2	0.08	0.08	53.99
3	3.70	0.61	112.63
4	0.12	0.12	191.36
5	12.49	1.32	291.67
6	0.62	0.26	413.12
7	38.85	0.29	555.44
8	2.99	1.44	720.76
9	235.49	0.51	905.50
10	138.76	0.52	1110.95
11	107.00	2.62	893.94

Table 7.4: Table of *Quality* per group size (K) for real data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system A

K	CPU-DePermute	GPU-DePermute	CLUMPP
2	$2.44E{-}08$	$2.44E{-}08$	$1.00E{+}00$
3	$2.07E{-}02$	$3.17E{-}03$	$8.30E{-}01$
4	$2.13E{-}07$	$2.13E{-}07$	$1.00E{+}00$
5	$2.13E{-}02$	$1.82E{-}03$	$7.62E{-}01$
6	$9.95E{-}04$	$1.87E{-}04$	$9.87E{-}01$
7	$4.97E{-}03$	$8.51E{-}04$	$9.21E{-}01$
8	$4.05E{-}04$	$8.13E{-}04$	$9.92E{-}01$
9	$4.27E{-}03$	$7.30E{-}04$	$9.04E{-}01$
10	$7.54E{-}03$	$9.09E{-}04$	$8.39E{-}01$
11	$1.04E{-}02$	$4.24E{-}04$	$8.16E{-}01$

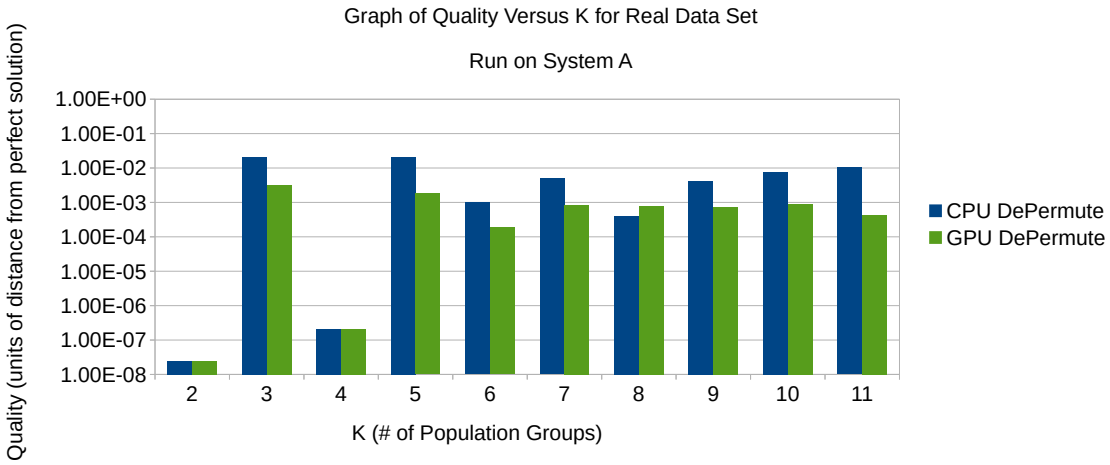


Figure 7.7: Graph of *Quality* versus (K) for real data testing of DePermute run on system A

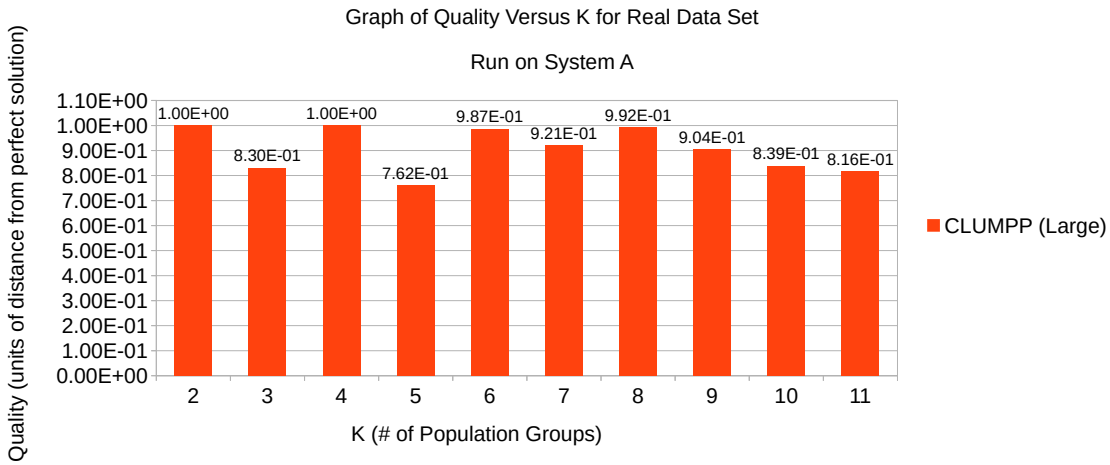


Figure 7.8: Graph of *Quality* versus (K) for real data testing of CLUMPP run on system A

7.3 System B

System B provides an interesting contrast to the other result sets because it was running the latest version of the Windows OS unlike the other two platforms (which both were running Linux).

Given the power of the CPU on system B one would have expected it to outperform system A, however the measured running time was significantly higher for all CPU based testing. This may have been due to software incompatibility, because the specifications of system B would otherwise indicate that the program should perform better and system B was the only machine which ran another OS.

7.3.1 Randomly Generated Data

Table 7.5: Table of *Time* per group size (K) for randomly generated data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system B

K	CPU-DePermute (seconds)	GPU-DePermute (seconds)	CLUMPP (seconds)
10	477.30	8.73	128.06
15	N/A	8.26	225.74
20	N/A	16.63	398.53

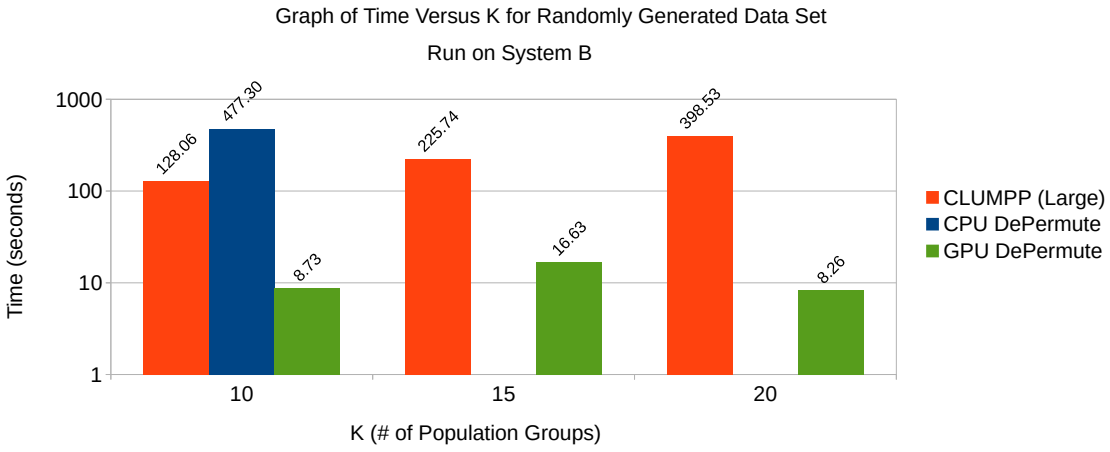


Figure 7.9: Graph of *Time* versus (K) for randomly generated data testing of CLUMPP and DePermute run on system B

7.3.2 Real Data

Table 7.6: Table of *Time* per group size (K) for real data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system B

K	CPU-DePermute (seconds)	GPU-DePermute (seconds)	CLUMPP (seconds)
2	2.15	2.48	31.58
3	44.27	4.70	65.84
4	3.99	4.32	112.48
5	143.22	6.82	171.30
6	9.66	6.61	242.36
7	394.10	8.06	325.74
8	29.01	9.52	421.59
9	2028.07	10.39	529.14
10	1159.38	11.03	651.29
11	985.47	9.86	525.23

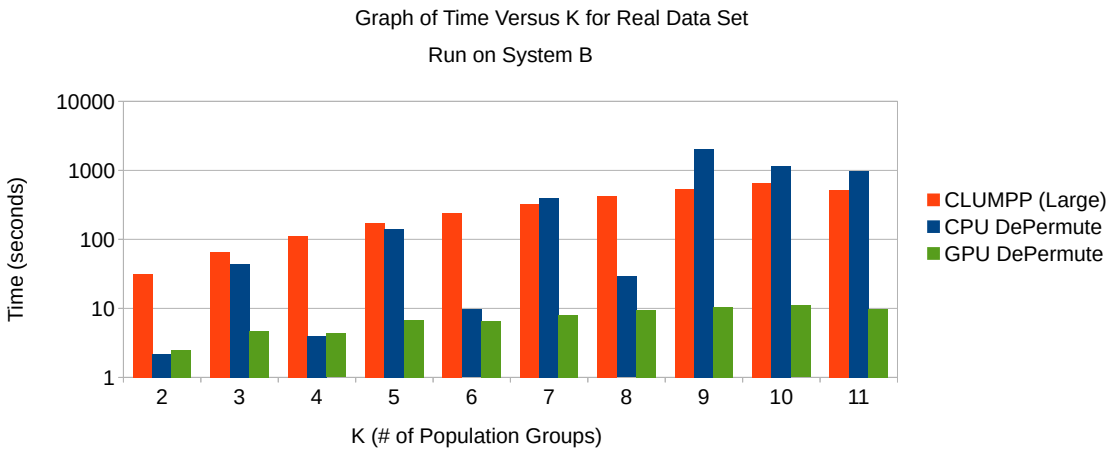


Figure 7.10: Graph of *Time* versus (K) for real data testing of CLUMPP and DePermute run on system B

7.4 System C

System C is the only instance where the high quality settings (i.e. greedy) for the CLUMPP algorithm were run. This is because System C was the first test conducted. After the test was conducted it was determined that this mode of the algorithm was significantly slower than any other program and did not provide a significant advantage in quality over the other setting for CLUMPP, therefore it wasn't tested on other systems.

GPU based testing was not conducted on System C, therefore the results only contain entries for CPU-DePermute and the two quality settings for which CLUMPP was run.

Note that the high quality mode of CLUMPP did not finish processing any of the randomly generated data sets within the cut off time and was too slow to finish the majority of the K values for the real data set.

DePermute achieved lower running times than CLUMPP except in the case of the $K = 20$ data set. Here we see that CLUMPP outperforms DePermute by approximately 100 seconds in each of the three runs. We note that the CLUMPP greedy algorithm has a higher constant factor, but appears to scale better than DePermute on a CPU at larger K.

DePermute performs particularly well at low K, therefore we propose that the poor scaling exhibited in this test is caused by DePermutes reliance on threads to perform searches in designated partitions. In the testing procedure the number of partitions was set to, at least, K in each test, because that is the smallest number of partitions which DePermute supports. When K exceeds the number of cores on a machine, then the threads are starved for processing time and I/O bottlenecks are also aggravated.

Finally note that solutions of identical quality were found by both settings on which CLUMPP was run. This may be indicative of very clear structure in the test data or of the nature of greedy algorithms in becoming stuck in local optima which are easily found.

7.4.1 Randomly Generated Data

Table 7.7: Table of *Time* per group size (K) for randomly generated data testing of CPU-DePermute and CLUMPP run on system C

K	CPU-DePermute (seconds)	CLUMPP (seconds)
10	47.23	111.51
15	175.70	243.04
20	488.46	429.72

7.4.2 Real Data

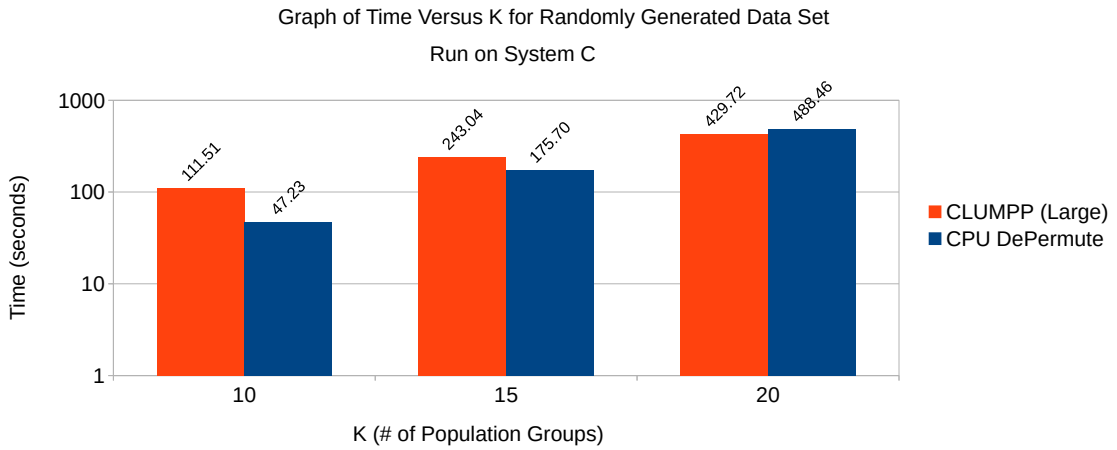


Figure 7.11: Graph of *Time* versus (K) for randomly generated data testing of CLUMPP and DePermute run on system C

Table 7.8: Table of *Time* per group size (K) for real data testing of CPU-DePermute, high quality settings for CLUMPP and CLUMPP run on system C

K	CPU-DePermute (seconds)	High Quality CLUMPP (seconds)	CLUMPP (seconds)
2	0.21	413.995	62.38
3	4.80	1863.23	131.11
4	0.32	9900.845	223.23
5	12.66	62314.96	335.75
6	1.42	N/A	474.14
7	41.46	N/A	637.43
8	3.45	N/A	826.97
9	235.34	N/A	1044.00
10	140.38	N/A	1281.71
11	108.98	N/A	1051.69

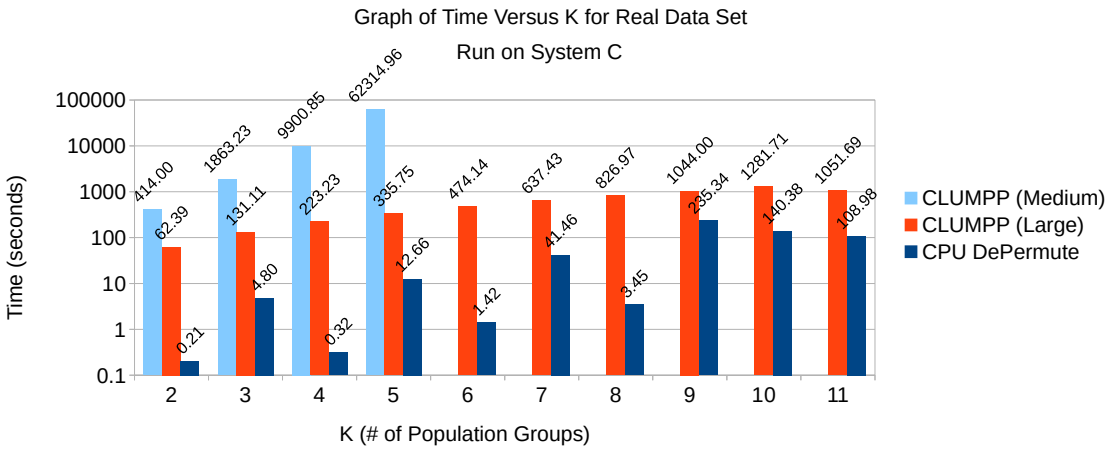


Figure 7.12: Graph of *Time* versus (K) for real data testing of CLUMPP and DePermute run on system C

Chapter 8

Conclusion

The field of computer science stands at a critical junction and requires a fundamental shift from sequential to parallel paradigms of computation. A more fundamental approach to the problem would benefit both industry and academic research. Efforts are being made in research to advance our understanding of parallel computation and to create a paradigm shift in the way industry and researchers reason about parallelism.

It has been established, through this and other research, that parallelism, when used effectively and given a problem which is amenable to it, usually allows one to solve larger problems within tighter time constraints.

A central tenet of this research is that there are commonalities in the algorithms and strategies for parallelism for metaheuristics which have previously not been taken advantage of. To this end this dissertation presents the Parallel Solvers model designed to create parallel metaheuristic algorithms to solve combinatorial problems. This model is used to implement an algorithm called DePermute which solves the Label Switching Problem (LSP).

We have established what impact various heuristics have on the number of iterations to find a sufficiently high quality solution. Importantly it has been established that the most effective parameter of the algorithm is the size of the initial population. Other parameters, such as mutation and elitism were found to have adverse effects on the number of iteration at high levels. The two parameters are most likely effective in practical tests because of the limitations on memory capacity and bandwidth which prohibit the use of very large populations.

Divisions of the search space as a heuristic was found to be ineffectual on its own, and had unpredictable effects on the number of iterations when used in conjunction with communication. However, both of these were found – through ad hoc testing and performance based tests – to be an important aspect in making effective use of parallel computating resources.

Implementations of DePermute using a CPU based computer, a GPU based computer and an FGPA hybrid computer are presented. The GPU based implementation uses the OpenCL library for GPU based acceleration and the FGPA implementation uses the Convey Platform as a means of accelerating x86 applications with customised hardware.

The GPU and CPU based prototypes were implemented and are available for use in research. The prototype programs were tested by measuring running time and quality of

result and contrasted with results achieved by the CLUMPP program on two settings (Greedy and Large K Greedy).

It was found that the CPU-DePermute consistently outperforms CLUMPP on fast settings when tested using the real data set. Achieving a 19.21 times speedup over CLUMPP on average. It was concluded that this was largely due to the simplicity of the underlying problems present in the real data set. It was found that GPU-DePermute achieved a 576.0 times speedup over CLUMPP on average by the same measurement.

In terms of quality of solution achieved it was found that DePermute fluctuates between runs and between problem sizes to a greater degree than CLUMPP. Furthermore it was found that the quality of result achieved by CLUMPP was highly consistent when run multiple times and across multiple problem sizes. Given that both programs were found to generate similar solutions and that the measure which each program uses to determine quality is slightly different a meaningful comparison between the two programs can't be drawn in terms of quality.

DePermute allows one to solve larger problems in less time with similar quality to CLUMPP. It demonstrates the success of the Parallel Solvers model in its application to the LSP. The massively parallel paradigm of computation has been found to be amenable to solving combinatorial problems in this instance and the model presented herein might also be effective in accelerating solvers for other combinatorial problems as well.

Appendix A

Extra Quality Results

A.1 System B

Table A.1: Table of *Quality* per group size (k) for randomly generated data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system B

K	CPU-DePermute	GPU-DePermute	CLUMPP
10	$1.00E{-}02$	$2.34E{-}03$	$2.34E{-}03$
15	–	$8.24E{-}04$	$8.24E{-}04$
20	–	$8.49E{-}04$	$8.49E{-}04$

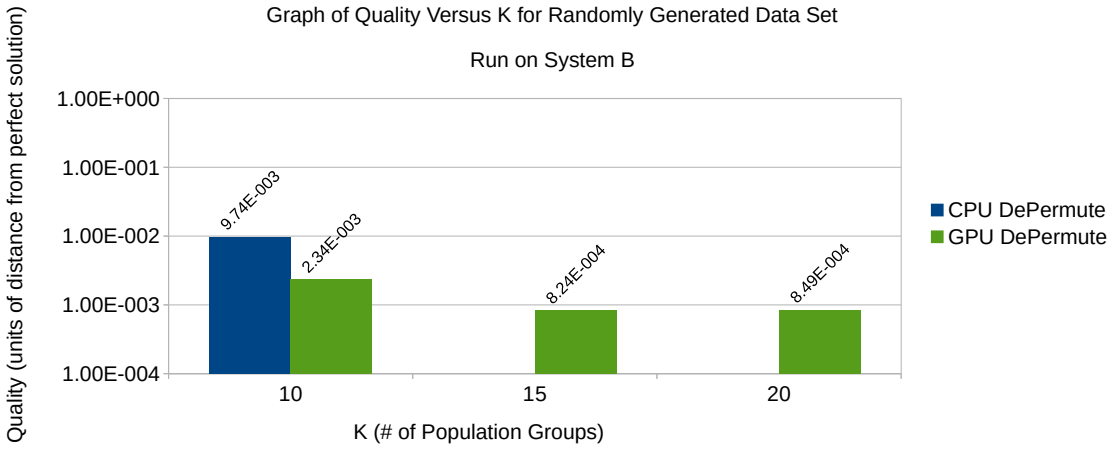


Figure A.1: Graph of *Quality* versus (k) for randomly generated data testing of DePermute run on system B

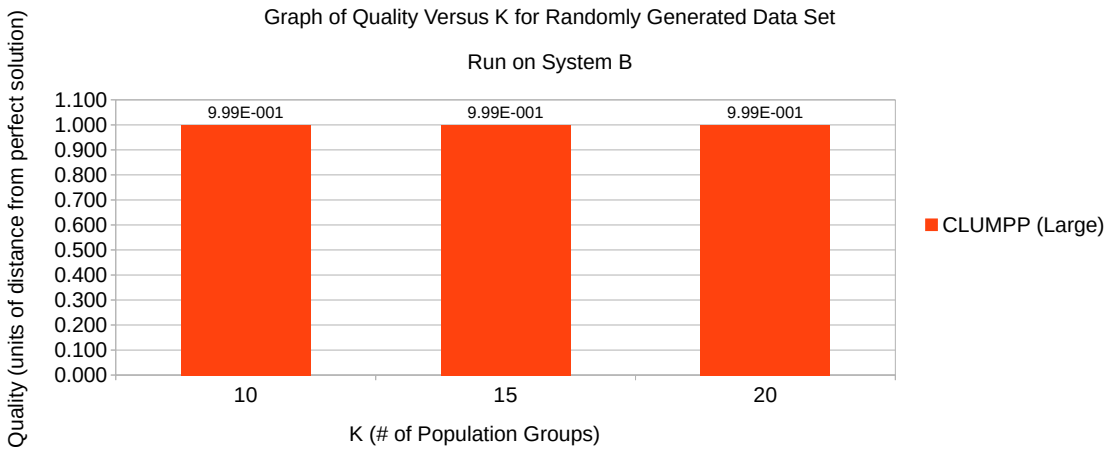


Figure A.2: Graph of *Quality* versus (k) for randomly generated data testing of CLUMPP run on system B

Table A.2: Table of *Quality* per group size (k) for real data testing of CPU-DePermute, GPU-DePermute and CLUMPP run on system B

K	CPU-DePermute	GPU-DePermute	CLUMPP
2	$2.44E{-}08$	$2.44E{-}08$	$1.00E{+}00$
3	$2.07E{-}02$	$2.84E{-}03$	$8.30E{-}01$
4	$2.13E{-}07$	$2.13E{-}07$	$1.00E{+}00$
5	$2.13E{-}02$	$2.16E{-}03$	$7.62E{-}01$
6	$9.95E{-}04$	$3.68E{-}04$	$9.87E{-}01$
7	$4.97E{-}03$	$6.89E{-}04$	$9.21E{-}01$
8	$4.05E{-}04$	$7.98E{-}04$	$9.92E{-}01$
9	$4.27E{-}03$	$5.46E{-}04$	$9.04E{-}01$
10	$7.54E{-}03$	$9.09E{-}04$	$8.39E{-}01$
11	$1.04E{-}02$	$6.29E{-}04$	$8.16E{-}01$

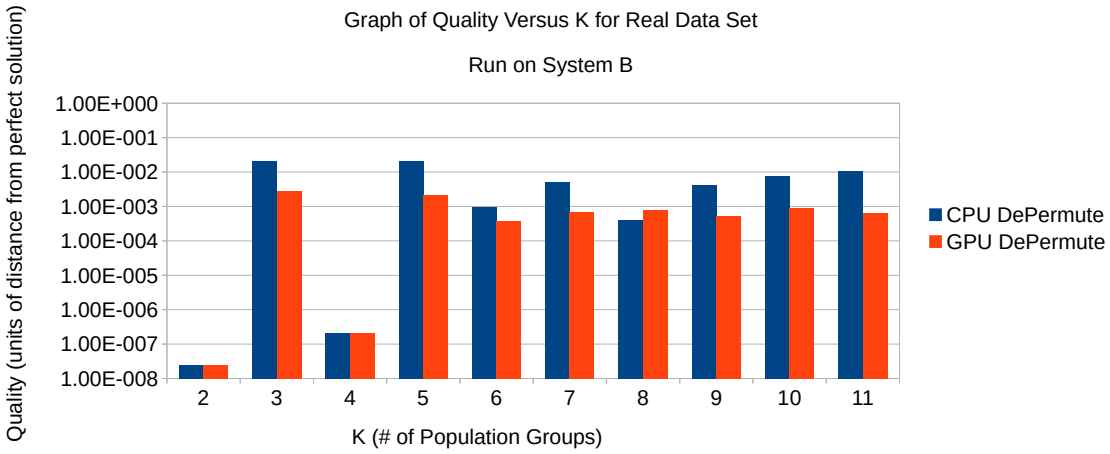


Figure A.3: Graph of *Quality* versus (k) for real data testing of DePermute run on system B

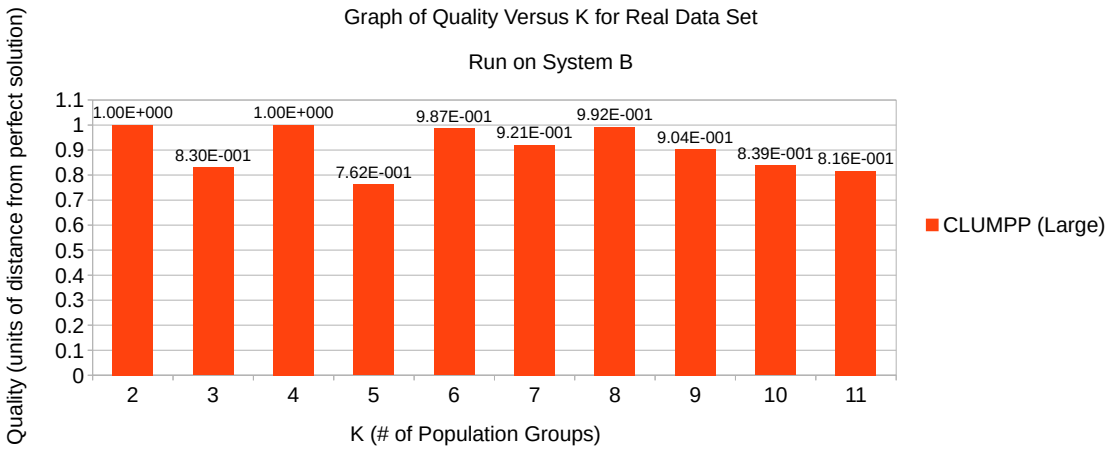


Figure A.4: Graph of *Quality* versus (k) for real data testing of CLUMPP run on system B

A.2 System C

Table A.3: Table of *Quality* per group size (k) for randomly generated data testing of CPU-DePermute and CLUMPP run on system C

K	CPU-DePermute	CLUMPP
10	$1.00E{-}02$	$9.99E{-}01$
15	$1.10E{-}02$	$9.99E{-}01$
20	$9.46E{-}03$	$9.99E{-}01$

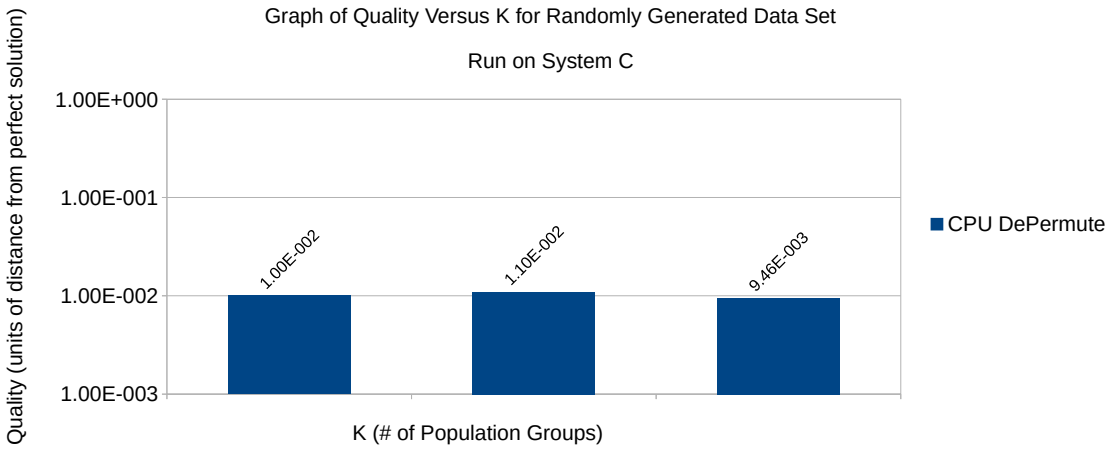


Figure A.5: Graph of *Quality* versus (k) for randomly generated data testing of DePermute run on system C

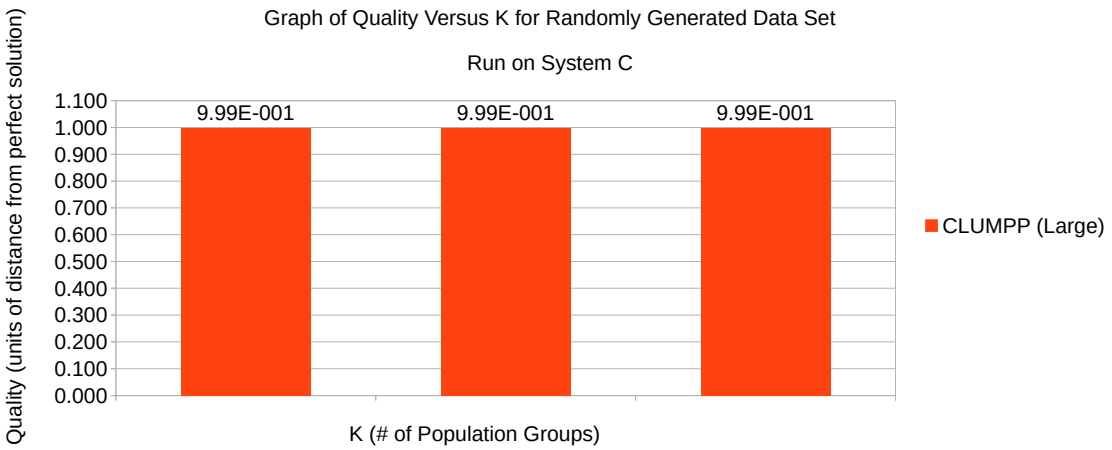


Figure A.6: Graph of *Quality* versus (k) for randomly generated data testing of CLUMPP run on system C

Table A.4: Table of *Quality* per group size (k) for real data testing of CPU-DePermute, high quality settings for CLUMPP and ordinary settings for CLUMPP run on system C

K	Run 1	Run 2	Average
2	$2.44E{-}08$	$1.00E{-}00$	$1.00E{+}00$
3	$2.07E{-}02$	$8.30E{-}01$	$8.30E{-}01$
4	$2.13E{-}07$	$1.00E{+}00$	$1.00E{+}00$
5	$2.13E{-}02$	$7.62E{-}01$	$7.62E{-}01$
6	$9.95E{-}04$	–	$9.87E{-}01$
7	$4.97E{-}03$	–	$9.21E{-}01$
8	$4.05E{-}04$	–	$9.92E{-}01$
9	$4.27E{-}03$	–	$9.04E{-}01$
10	$7.54E{-}03$	–	$8.39E{-}01$
11	$1.04E{-}02$	–	$8.16E{-}01$

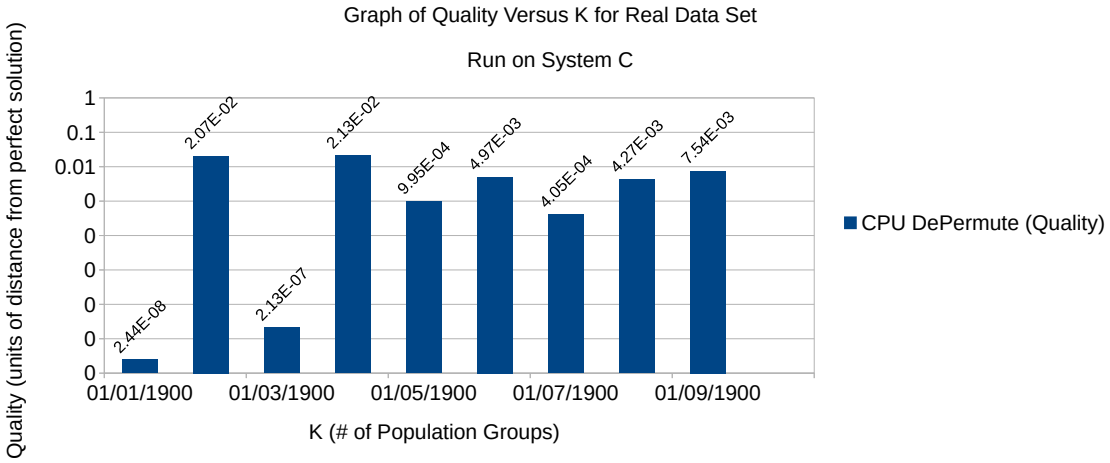


Figure A.7: Graph of *Quality* versus (k) for real data testing of DePermute run on system C

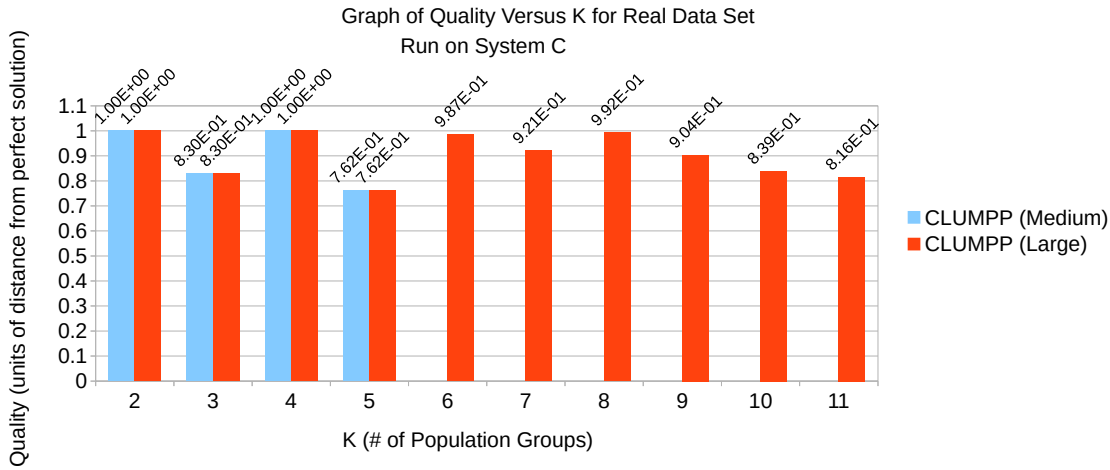


Figure A.8: Graph of *Quality* versus (k) for real data testing of CLUMPP run on system C

Appendix B

Extra Sample Information

The suspected population groups, the count of members who believe that they are representative of that group and the population group codes for all members of the sample data set are listed as follows:

Population Group Description	Code	Count in Experiment
African Caribbean in Barbados	ACB	96
African Ancestry in Southwest US	ASW	66
Bengali in Bangladesh	BEB	86
South Eastern Bantu Speakers from the Soweto Region	BSO	94
Chinese Dai in Xishuangbanna, Chin	CDX	99
Han Chinese in Bejing, China	CHB	103
Southern Han Chinese, China	CHS	108
Colombian in Medellin, Colombia	CLM	94
Coloured living in Wellington	WCOL	40
Coloured living in Colesberg	COL	28
Esan in Nigeria	ESN	99
Utah residents with Northern and Western European ancestry	EUR	207
Finnish in Finland	FIN	99
British in England and Scotland	GBR	92
Gujarati Indian in Houston,TX	GIH	106
Gambian in Western Division, The Gambia	GWD	113
Guinea, Ghana in Kigali	GuiGhanaKgal	14
Iberian populations in Spain	IBS	107
Indian Telugu in the UK	ITU	103
Japanese in Tokyo, Japan	JPT	104

Kinh in Ho Chi Minh City, Vietnam	KHV	101
Khoe San	KS	39
Khomani San in South Africa	Khomani	39
Khwe Speaking San	Khwe	17
Luhya in Webuye, Kenya	LWK	101
Mende in Sierra Leone	MSL	85
Mexican Ancestry in Los Angeles, California	MXL	67
Nama San in Namibia	Nama	20
Peruvian in Lima, Peru	PEL	86
Punjabi in Lahore,Pakistan	PJL	96
Puerto Rican in Puerto Rico	PUR	105
South Eastern Bantu Speakers	SEB	20
Sotho	SOT	8
Malay (Singapore Sequencing Malay Project)	SSMP	97
Sri Lankan Tamil in the UK	STU	103
South Western Bantu Speakers	SWB	12
Toscani in Italia	TSI	108
Xhosa	XHO	8
Unknown	XXX	1
Xun San	Xun	19
Yoruba in Ibadan, Nigeria	YRI	109

Appendix C

Scripts & Test Programs

C.1 Random Data Generation

Listing C.1: Random Data Generation Bash Script

```
# This script generates data according to the parameters stored in the
    ProblemParams.sh file.

source ProblemParams.sh

mkdir $TEST_DATA_DIR

for (( k=$K_START; k <= $K_END; k=$k + $K_STEP ))
do
    noise $k
    for (( r=$R_START; r <= $R_END; r=$r + $R_STEP ))
    do
        for (( repeat=0; repeat != $REPEAT; repeat++ ))
        do
            matrixName $k $r $repeat
            ./GenerateTestMatrices $k $r $INDIVIDUALS $NOISE >
    $TEST_DATA_DIR/$MAT_FILE_NAME
            python SplitMats.py $TEST_DATA_DIR/${MAT_FILE_NAME}_ <
    $TEST_DATA_DIR/$MAT_FILE_NAME
        done
    done
done
```

Listing C.2: Random Data Problem Size Settings

```
# Parameters for tests pertaining to random data set generation

# ===Experimental Parameters===
```

```
# The name of the directory to do the tests in
DIR=Random-Test-Dir
# Aggregated results files for each program
CLUMPP_AGGREGATED_RESULT=$DIR/CLUMPP.out
CPU_DEPERMUTE_AGGREGATED_RESULT=$DIR/CPU-DePermute.out
GPU_DEPERMUTE_AGGREGATED_RESULT=$DIR/GPU-DePermute.out
AVERAGED_RESULTS=$DIR/AverageResults.out
# The range of K values to be tested
K_START=10
K_END=20
K_STEP=5
# The value of R in all tests
R=20
# The number of individuals in the study
M=500
# The number of times that the experiment should be repeated
REPEAT=3

# ===Functions====

# The amount of noise to super impose on the data
function noise # ( k )
{
    N=0.01
}

# Produces the path to the base matrix file for the given K and repeat number
# Int Int -> String
function matrixFile # ( k, repeatNumber )
{
    K=$1
    REPEAT_NUMBER=$2

    MAT_FILE=$DIR/Matrices-$K-$REPEAT_NUMBER
}

source SharedSetup.sh
```

Listing C.3: Script to Split Matrices Into Separate Files as is Appropriate for DePermute

```
# ===Standard Depedencies===

import sys

# ===Procedure===

# Setup variables for reading in matrices
# Expects the name of the file to put the matrices into after splitting them.
```

```
# Reads the matrices from StdIn in new line seperated matrices in space (col) newline (row) seperated
# format
lines    = ""
baseFile = sys.argv[1]
l        = 0

# Parse the matrix file piped to StdIn
for line in sys.stdin:
   # In the case that we hit a newline, we've finished parsing a matrix and move to the next
   if line == "\n":
      out = open ( baseFile + str ( l ), "w" )
      l += 1
      out.write  ( lines )
      out.close  ()
      lines = ""
   # Otherwise we add the next line to the current "matrix" as a string
   else:
      lines = lines + line
```

Listing C.4: Program to Generate Random Data for Testing Purposes

```
module Main
    where

-- ===Standard Dependencies===

import Control.Monad.State.Lazy
import System.Environment
import System.Random

-- ===Non Stanadard Dependencies===

import Matrix

-- ===Functions===

-- Produces a collection of test matrices given K, R, M and N where
--  - K is width
--  - R is the number of runs
--  - M is the number of individuals
--  - N is the noise level in the data
-- Usage ./GenerateTestMatrices <K> <R> <M> <N>
main :: IO ()
main = do
  rnd  <- getStdGen
  args <- getArgs
```

```
  if length args /= 4 then
      printUsage
  else do
    let (sk : sr : sm : sn : []) = args
    let k = read sk
    let r = read sr
    let m = read sm
    let n = read sn
    let matShuffleIndPairs = evalState ( generateMatrices k r m n ) rnd
    mapM_ ( \ (mat, _)  -> printRowMajor mat ) matShuffleIndPairs
    mapM_ ( \ (_, inds) -> print inds ) matShuffleIndPairs

-- Prints how this program should be used to the user
printUsage :: IO ()
printUsage = putStrLn $ "Usage:\n ./GenerateTestMatrices <K> <R> <M> <N>"
```

Listing C.5: Supporting Library of Monadic Stochastic Functions

```
module Stochastic
    where

-- ===Standard Dependencies===

import Control.Monad.State.Lazy
import System.Random
import Data.Word

-- ===Functions===

-- Produces n random numbers in the state monad
nRandomsST :: (RandomGen g, Random a) => Int -> State g [a]
nRandomsST n = mapM ( \ x -> randomST ) [1..n]

-- Produces n random numbers in the given range in the state monad
nRandomsRST :: (RandomGen g, Random a) => Int -> (a, a) -> State g [a]
nRandomsRST n range = mapM ( \ x -> randomRST range ) [1..n]

-- Produces a random variable in the state monad
randomST :: (RandomGen g, Random a) => State g a
randomST = state random

-- Produces a random variable in the given range in the state monad
randomRST :: (RandomGen g, Random a) => (a, a) -> State g a
randomRST range = state $ randomR range
```

Listing C.6: Supporting Library of Functions Which Operate on Matrices

```
module Matrix
```

```
    where

-- ===Standard Dependencies===

import qualified Data.List as L
import Control.Monad.State.Lazy
import Control.Monad
import System.Random
import Data.Array

-- ===Non Standard Dependencies===

import Stochastic

-- ===Data Defenitions===

-- RowMatrix is [[Float]]
-- interp. A representation of a matrix as a list of lists in Row major format
type RowMatrix = [[Float]]

-- ColMatrix is Array Float [Float]
-- interp. A representation of a matrix as an array of lists in column major
    format
type ColMatrix = Array Int [Float]

-- ===Functions===

-- Prints a row major matrix in row major format with space separated values
printRowMajor :: RowMatrix -> IO ()
printRowMajor mat = do
  let rowToString x = concat $ L.intersperse " " $ map show x
  mapM_ ( \ x -> putStrLn ( rowToString x ) ) mat
  putStr "\n"

-- Produces m random, similar, r by k, matrices (with noise induced @ n)
generateMatrices :: Int -> Int -> Int -> Float -> State StdGen [(RowMatrix,
    [Int])]
generateMatrices k r m n = do
  mats         <- nRandomMatrices m k r n
  shuffledMats <- shuffleColumns k mats
  return shuffledMats

-- Produces one random matrix and t - 1 copies of that matrix with n noise
    imposed on the copies
nRandomMatrices :: Int -> Int -> Int -> Float -> State StdGen [RowMatrix]
nRandomMatrices rows cols t n = do
  mat         <- randomMatrix rows cols
  let copies   = replicate ( t - 1 ) mat
```

```
  noiseCopies <- mapM ( flip induceNoise $ n ) copies
  return $ mat : noiseCopies

-- Produces a random RowMatrix where the rows are normalised
randomMatrix :: Int -> Int -> State StdGen RowMatrix
randomMatrix rows cols = do
  rnds <- nRandomsRST ( rows * cols ) (0.0, 1.0)
  return $ take rows $ map (normalise . take cols) $ iterate (drop cols) rnds

-- Induces additive noise @ N noise level to each row of the given matrix
induceNoise :: RowMatrix -> Float -> State StdGen RowMatrix
induceNoise mat n = do
  let l        = length ( mat !! 0 )
  let scaled   = n / ( fromIntegral l )
  let induce x = do
        rnds <- nRandomsRST l (0.0, scaled)
        return ( normalise $ map (uncurry (+)) $ zip x rnds )
  mapM induce mat

-- Normalises the given list
normalise :: [Float] -> [Float]
normalise xs =
    let s = sum xs
    in map (/s) xs

-- Shuffles the columns of the given RowMatrices
shuffleColumns :: Int -> [RowMatrix] -> State StdGen [(RowMatrix, [Int])]
shuffleColumns cols mats = do
  shuffled <- mapM ( colShuffle cols ) $ map rowToColMat mats
  return $ map ( \ (mat, inds) -> (colToRowMat mat, inds) ) shuffled

-- Shuffles the columns in the given ColMatrix randomly
colShuffle :: Int -> ColMatrix -> State StdGen (ColMatrix, [Int])
colShuffle cols mat = do
  indices <- knuthShuffle [0..cols - 1]
  let shuffled = listArray (0, cols - 1) $ map (mat!) indices
  return (shuffled, indices)

-- Shuffles the elements in the given list
knuthShuffle :: Eq a => [a] -> State StdGen [a]
knuthShuffle [] = return []
knuthShuffle xs = do
  pos   <- randomRST (0, length xs - 1)
  let x  = xs !! pos
  rest  <- knuthShuffle ( L.delete x xs )
  return $ x : rest

-- Converts the given RowMatrix to a ColMatrix
```

```
rowToColMat :: RowMatrix -> ColMatrix
rowToColMat mat =
    let (rows, cols) = (length ( mat !! 0 ), length mat)
    in listArray (0, cols - 1) $ L.transpose mat

-- Converts the given ColMatrix to a RowMatrix
colToRowMat :: ColMatrix -> RowMatrix
colToRowMat mat =
    L.transpose $ elems mat
```

C.2 Convergence Based Tests

C.2.1 Shared Scripts

Listing C.7: Generates a Line for Every Desired Configuration for DePermute

```
# This script generates the parameters for a number of runs of DePermute and the
    headers of the CSV
# results files for aggregation.
#
# It expects a flag for the parameter which should be tuned and values for the
    start, end and step
# for the range of values it should be changed through.
#
# This script works by shadowing the function to generate the needed parameter,
    using the afore
# defined functions when they haven't been shadowed.

source ProblemParams.sh

# Produces the nth line of the given file
# Int String -> String
function nthLine # ( n, filePath )
{
    N=$1
    FILE_PATH=$2

    NTH_LINE=`sed -n "${N}{p;q;}" $FILE_PATH`
}

# Produces the appropirate division count given the problem width (K)
# Int -> Int
function divCount # ( k )
{
    K=$1

    nthLine $K $DIV_FILE
    DIVISION_COUNT=$NTH_LINE
```

```
}

# Produces the appropriate population size given K and the number of divisions
# Int Int -> Int
function popSize # ( k, divs )
{
    K=$1
    DIVS=$2

    nthLine $K $POP_FILE
    POP_SIZE=$(($NTH_LINE / $DIVS))
}

# Produces the appropriate elitist count for the given value of K and POP_SIZE
# Int Int -> Int
function eliteCount # ( k, popSize )
{
    K=$1
    POP=$2

    nthLine $K $ELITE_FILE
    ELITE_COUNT=`python -c "print int ( $NTH_LINE * $POP )"`
}

# Produces the appropriate mutatiton rate for the given value of K
# Int -> Int
function mutationRate # ( k )
{
    K=$1

    nthLine $K $MUT_FILE
    MUTATION_RATE=$NTH_LINE
}

# Produces the appropriate proportion of the population to be communicated given
    a value of K and
# the popSize
# Int Int -> Int
function commCount # ( k, popSize )
{
    K=$1
    POP=$2

    nthLine $K $COMM_PROP
    COMM=`python -c "print int ( $NTH_LINE * $POP )"`
}

# The string to replace with the current repeat number
```

```
77 REPEAT_PLACE_HOLDER=Repeat#

79 # Produces the parameters to run DePermute with a placeholder of [Repeat#] for
       the number of this repeat
   # Int -> String
81 function depermuteParams # ( k, r, repeatNumber )
   {
83     K=$1
       R=$2
85     REPEAT_NUMBER=$3

87     baseMatrixName $K $R
       divCount       $K
89     popSize        $K $DIVISION_COUNT
       eliteCount     $K $POP_SIZE
91     mutationRate   $K
       commCount      $K $POP_SIZE
93
       DEPERMUTE_PARAMS="-i
       ${TEST_DATA_DIR}/${MAT_FILE_NAME}-${REPEAT_PLACE_HOLDER}_ -e $ELITE_COUNT -r
       $R -p $POP_SIZE -c $DIVISION_COUNT -k $K -x $INDIVIDUALS -t 0.1 -m
       $MUTATION_RATE -s 100 -y 0 -n $COMM"
95 }

97 # Generates the parameters needed for the DePermute program to run for the
       required experimental
   # parameters
99 # IO ()
   function saveDePermuteParamStrings # ()
101 {
       for (( k=$K_START; k <= $K_END; k++ )); do
103        for (( r=$R_START; r <= $R_END; r++ )); do
               resetIndependantVar
105            startGeneratingXVals
               while [ "$STOP" = false ]; do
107                # I think that the problem is that I need to float the matrix
       file out of this
                   depermuteParams $k $r $repeatCount
109                for (( repeatCount=0; repeatCount < $REPEAT; repeatCount++ )); do

       FINAL_PARAMS=${DEPERMUTE_PARAMS//$REPEAT_PLACE_HOLDER/$repeatCount}
111                    echo $FINAL_PARAMS
                   done
113            done
           done
115    done
   }
117
```

```
# Flags that the generation of the independant variable should stop
# IO ()
function stopGeneratingXVals # ()
{
    STOP=true
}

# Flags that the generation of the independant variable should continue again
# IO ()
function startGeneratingXVals # ()
{
    STOP=false
}

# Flags that the generation of the independant variable should reset
# IO ()
function resetIndependantVar # ()
{
    RESET=true
}

# Flags that the generation of the independant variable shouldn't reset
# IO ()
function dontResetIndependantVar # ()
{
    RESET=false
}

RANGE_START=$2
RANGE_END=$3
RANGE_STEP=$4
STOP=false
DELTA=0.0001

if [ -z "$1" ]; then
    echo "Please provide the name of the parameter which you want to permute"
    exit 1
elif [ "$1" = "Elitism" ]; then
    function eliteCount # ( k, popSize )
    {
        if [ "$RESET" = true ]; then
            ELITISM_PROP=`python -c "print $RANGE_START - $RANGE_STEP"`
            dontResetIndependantVar
        fi
        ELITISM_PROP=`python -c "print $ELITISM_PROP + $RANGE_STEP"`
        ELITE_COUNT=`python -c "print int ( $ELITISM_PROP * $2 )"`
        if [ `echo | gawk "{if ($ELITISM_PROP > ($RANGE_END - $DELTA)) print
    (1); else print (0)}"` -eq 1 ]; then
```

```
165             stopGeneratingXVals
            fi
167     }
        saveDePermuteParamStrings
169 elif [ "$1" = "Population" ]; then
        function popSize # ( k )
171     {
            echo "Unimplemented: See Test1.sh for implementation details."
173         exit 1
            # Start of impl...
175         if [ "$RESET" = true ]; then
                POP_SIZE=$(($RANGE_START - $RANGE_STEP))
177             dontResetIndependantVar
            fi
179         POP_SIZE=$(($POP_SIZE + $RANGE_STEP))
            if [ $POP_SIZE -eq $RANGE_END ]; then
181             stopGeneratingXVals
            fi
183     }
        saveDePermuteParamStrings
185 elif [ "$1" = "Mutation" ]; then
        function mutationRate # ( k )
187     {
            if [ "$RESET" = true ]; then
189             MUTATION_RATE=`python -c "print $RANGE_START - $RANGE_STEP"`
                dontResetIndependantVar
191         fi
            MUTATION_RATE=`python -c "print $MUTATION_RATE + $RANGE_STEP"`
193         if [ `echo | gawk "{if ($MUTATION_RATE > ($RANGE_END - $DELTA)) print
    (1); else print (0)}"` -eq 1 ]; then
                stopGeneratingXVals
195         fi
        }
197     saveDePermuteParamStrings
    elif [ "$1" = "Divisions" ]; then
199     function divCount # ( k )
        {
201         if [ "$RESET" = true ]; then
                DIVISION_EXPONENT=$(($RANGE_START - $RANGE_STEP))
203             dontResetIndependantVar
            fi
205         DIVISION_EXPONENT=$(($DIVISION_EXPONENT + $RANGE_STEP))
            DIVISION_COUNT=`python -c "print $1**$DIVISION_EXPONENT"`
207         if [ `echo | gawk "{if ($DIVISION_EXPONENT > ($RANGE_END - $DELTA))
    print (1); else print (0)}"` -eq 1 ]; then
                stopGeneratingXVals
209         fi
        }
```

```
    saveDePermuteParamStrings
elif [ "$1" = "Communication" ]; then
    function commCount # ( k, popSize )
    {
        if [ "$RESET" = true ]; then
            COMM_PROP=`python -c "print $RANGE_START - $RANGE_STEP"`
            dontResetIndependantVar
        fi
        COMM_PROP=`python -c "print $COMM_PROP + $RANGE_STEP"`
        COMM=`python -c "print int ( $COMM_PROP * $2 )"`
        if [ `echo | gawk "{if ($COMM_PROP > ($RANGE_END - $DELTA)) print (1);
    else print (0)}"` -eq 1 ]; then
            stopGeneratingXVals
        fi
    }
    saveDePermuteParamStrings
else
    echo "Invalid indipendant variable please supply either Elitism, Population,
    Mutation, Divisions or Communication"
    exit 1
fi

# Write the csv heading to a file

source TestParams.sh

GENERATED=$RANGE_START
HEADING="${RANGE_START}"
STOP=false
while [ "$STOP" = false ]; do
    GENERATED=`python -c "print $GENERATED + $RANGE_STEP"`
    HEADING="${HEADING},${GENERATED}"
    if [ `echo | gawk "{if ($GENERATED > ($RANGE_END - $DELTA)) print (1); else
    print (0)}"` -eq 1 ]; then
        STOP=TRUE
    fi
done
echo $HEADING > $CSV_HEADING_FILE
```

Listing C.8: Runs the Program for the Given Set of Parameters

```
# This script runs, for each of the parameters piped to it, the CPUDepermute
    program storing logging
# data in one file and results in another as defined in the settings below.

source PBSArrayFallBack.sh
source TestParams.sh
```

```
mkdir -p $RESULTS_DIR

RESULTS_FILE_BASE_PATH=$RESULTS_DIR/$RESULT_FILE_BASE_NAME
LOG_FILE_BASE_PATH=$RESULTS_DIR/$LOG_FILE_BASE_NAME
PROC_COUNT=0

# Non cluster run?
if [ -z "$PBS_ARRAYID" ]; then
    PBS_ARRAYID=0
    CONCURRENT=1
fi

while read paramLine; do
    if (( $PROC_COUNT % $CONCURRENT == $PBS_ARRAYID )); then
        ./CPUDePermute $paramLine 2> $LOG_FILE_BASE_PATH${PROC_COUNT} 1>
    $RESULTS_FILE_BASE_PATH${PROC_COUNT}
    fi
    PROC_COUNT=$(($PROC_COUNT + 1))
done
```

C.2.2 Test 1

Listing C.9: Test 1 script

```
#!/bin/bash
#PBS -N Convergence
#PBS -l nodes=1:ppn=1,walltime=4:30:00,mem=2GB
#PBS -q WitsLong
#PBS -o /home/edward/Test1.log
#PBS -t 0-191
CONCURRENT=192

# The testing procedure for test one (note that this now includes the same
    functionality as the AggregateAverages script)

# ===Experimental Setup===

# Sub directory to run the test in
DIR=Test1
# [Start, End] range for K
K_START=9
K_END=10
# [Start, End] range for R
R_START=10
R_END=15
# The number of times that the test for each size should be repeated for
    averaging
REPEAT=2
```

```
# Individuals in the study
M=100
# Noise level in the simulated output of ADMIXTURE
N=0.01

# ===Supporting Functions===

# Produces the factorial of the given number
# Float -> Float
function fact # ( n )
{
    acc=1

    for ((l_fact=1; l_fact<=$1; l_fact++));
    do
        acc=$acc*$l_fact
    done

    FACT_RES=$acc
}

# Produces the given number raised to the given exponent
# Float -> Float
function pow # ( b, e )
{
    acc=1

    for ((l_pow=1; l_pow<=$2; l_pow++));
    do
        acc=$acc*$1
    done

   POW_RES=$acc
}

# The maximum number of the population to generate up to
# Int Int -> Int
function maxPop # ( K, R )
{
    case $1 in
        5)  POP_END=100000;;
        6)  POP_END=100000;;
        7)  POP_END=1000000;;
        8)  POP_END=1000000;;
        9)  POP_END=10000000;;
        10) POP_END=10000000;;
    esac
}
```

```
# The minimum proportion of the population to generate from
# Int Int -> Int
function minPop # ( K, R )
{
    case $1 in
        5)  POP_START=25;;
        6)  POP_START=2500;;
        7)  POP_START=25000;;
        8)  POP_START=250000;;
        9)  POP_START=250000;;
        10) POP_START=250000;;
    esac
}

# ===Main Program Start===

# Create test sub directory
mkdir $DIR
PROC_COUNT=0
# For each value of K in the decided range
for ((l=$K_START; l<=$K_END; l++));
do
    # For each value of R in the decided range
    for ((L=$R_START; L<=$R_END; L++));
    do
        maxPop $l $L
        minPop $l $L
        POP_INC=$POP_START
        # For increasing sizes of population up to the amount we specified as max
        for ((ll=$POP_START; ll<=$POP_END;ll=$ll+$POP_INC));
        do
            # Run the experiment three times to take the average of the attempts
            for ((LL=0; LL<$REPEAT; LL++));
            do
                if (( $PROC_COUNT % $CONCURRENT == $PBS_ARRAYID ))
                then
                    # Create the initial permuted matrices file
                    MAT_FILE=$DIR/Matrices-$l-$L-$M-$ll-$LL
                    ./GenerateTestMatrices.exe $l $L $M $N > $MAT_FILE
                    # Separate the permuted matrices
                    python SplitMats.py ${MAT_FILE}_ < $MAT_FILE
                    # Run DePermute on the file
                    echo "./DePermute.exe -i ${MAT_FILE}_ -e 0 -r $L -p $ll -c 1
    -k $l -x $M -t 0.01 -m 0.000001 -s 200 -y 0 -n 0 >
    $DIR/Results-$l-$L-$M-$ll-$LL.out"
                    ./DePermute.exe -i ${MAT_FILE}_ -e 0 -r $L -p $ll -c 1 -k $l
    -x $M -t 0.01 -m 0.000001 -s 200 -y 0 -n 0 > $DIR/Results-$l-$L-$M-$ll-$LL.out
```

```
                    # Aggregate iteration count results for this problem-population combination
                    cut -d" " -f2 $DIR/Results-$l-$L-$M-$ll-$LL.out | head -1 >> $DIR/Results-$l-$L-$M-$ll.out
                fi
                PROC_COUNT=$(($PROC_COUNT + 1))
            done
            if (($ll / ( 4 * $POP_INC ) == 1));
            then
                POP_INC=$(($POP_INC * 10))
                ll=0
            fi
        done
    done
done
```

C.2.3 Test 6

Listing C.10: Test 6 script

```
# The testing procedure for test Six

# ===Experimental Setup===

# Sub directory to run the test in
DIR=Test6
# [Start, End] range for K
K_START=5
K_END=20
# [Start, End] range for R
R_START=10
R_END=20
# The number of times that the test for each size should be repeated for averaging
REPEAT=20
# Individuals in the study
M=100
# Noise level in the simulated output of ADMIXTURE
N=0.01
# The number of concurrent processes to handle
CONCURRENT=4

# ===Main Program Start===

# Create test sub directory
rm -rf $DIR
mkdir $DIR
PROC_COUNT=0
```

```
# For each value of K in the decided range
for ((l=$K_START; l<$K_END; l++));
do
    # For each value of R in the decided range
    for ((L=$R_START; L<$R_END; L++));
    do
        # Run the experiment three times to take the average of the attempts
        for ((ll=0; ll<$REPEAT; ll++));
        do
            PROC_COUNT=`expr $PROC_COUNT + 1`
            (
                # Create the initial permuted matrices file
                MAT_FILE=$DIR/Matrices-$l-$L-$M-$ll
                ./GenerateTestMatrices.exe $l $L $M $N > $MAT_FILE
                # Separate the permuted matrices
                python SplitMats.py ${MAT_FILE}_ < $MAT_FILE
                # Run DePermute on the file
                echo "./CountTrivial.exe -i ${MAT_FILE}_ -e 0 -r $L -p 1 -c 1 -k $l -x $M -t 0.01 -m 0.00001 -s 200 -y 0 -n 0 > $DIR/Results-$l-$L-$M-$ll.out"
                ./CountTrivial.exe -i ${MAT_FILE}_ -e 0 -r $L -p 1 -c 1 -k $l -x $M -t 0.01 -m 0.00001 -s 200 -y 0 -n 0 > $DIR/Results-$l-$L-$M-$ll.out
                # Aggregate iteration count results for this problem-population combination
                cut -d" " -f2 $DIR/Results-$l-$L-$M-$ll.out | head -1 >> $DIR/Results-$l-$L-$M.out
            ) &
            if (($PROC_COUNT % CONCURRENT == 0));
            then
                wait
            fi
        done
        # Aggregate problem sizes
        echo "Problem size (R, K): ($L, $l)" >> $Dir/Results-$l-$L.out
        cat $DIR/Results-$l-$L-$M.out >> $Dir/Results-$l-$L.out
    done
done
```

C.3 Time Based Testing

Listing C.11: A script which is shared between both the process for testing using real and the process for using random data

```
# Shared parameters and functions for Time based testing

# ===Parameters===

# Path to the file containing optimal population counts
```

```
POP_FILE=PopCount.ini
# Path to the file containing the number of divisions to use
DIV_FILE=Divs.ini
# Path to the file containing mutation rates
MUT_FILE=Mutations.ini
# Path to the file containing elitism rates
ELITE_FILE=Elitism.ini
# Path to the file containing the proportion of the population which ought to be
    communicated
COMM_PROP=Comms.ini

# ===Functions===

# Produces the nth line of the given file
# Int String -> String
function nthLine # ( n, filePath )
{
    N=$1
    FILE_PATH=$2

    NTH_LINE=`sed -n "${N}{p;q;}" $FILE_PATH`
}

# Produces the appropirate division count given the problem width (K)
# Int -> Int
function divCount # ( k )
{
    K=$1

    nthLine $K $DIV_FILE
    DIVISION_COUNT=$NTH_LINE
}

# Produces the appropriate population size given K and the number of divisions
# Int Int -> Int
function popSize # ( k, divs )
{
    K=$1
    DIVS=$2

    nthLine $K $POP_FILE
    POP_SIZE=$(($NTH_LINE / $DIVS))
}

# Produces the appropriate elitist count for the given value of K and POP_SIZE
# Int Int -> Int
function eliteCount # ( k, popSize )
{
```

```
53      K=$1
        POP=$2
55
        nthLine $K $ELITE_FILE
57      ELITE_COUNT=`python -c "print int ( ( $NTH_LINE * $POP ) / $K )"`
}
59
# Produces the appropriate mutatiton rate for the given value of K
61 # Int -> Int
function mutationRate # ( k )
63 {
        K=$1
65
        nthLine $K $MUT_FILE
67      MUTATION_RATE=$NTH_LINE
}
69
# Produces the appropriate proportion of the population to be communicated given
    a value of K and
71 # the popSize
# Int Int -> Int
73 function commCount # ( k, popSize )
{
75      K=$1
        POP=$2
77
        nthLine $K $COMM_PROP
79      COMM=`python -c "print int ( ( $NTH_LINE * $POP ) / $K )"`
}
81
# Produces the path of the results file for runs of CLUMPP
83 # String
function clumppResultsFile # ( k, repeatNumber )
85 {
        K=$1
87      REPEAT_NUMBER=$2

89      CLUMPP_RESULTS_FILE=$DIR/Results-Clumpp-$K-$REPEAT_NUMBER.out
}
91
# Produces the path of the log file for runs of CLUMPP
93 # String
function clumppLogFile # ( k, repeatNumber )
95 {
        K=$1
97      REPEAT_NUMBER=$2

99      CLUMPP_LOG_FILE=$DIR/Log-Clumpp-$K-$REPEAT_NUMBER.out
```

```
}

# Produces the paramaters to run CLUMPP
# Int Int -> String
function clumppParams # ( k, repeatNumber )
{
    K=$1
    REPEAT_NUMBER=$2

    matrixFile $K $REPEAT_NUMBER

    CLUMPP_PARAMS="paramfile -p $MAT_FILE.clumpp -o
    $DIR/Perm-Clumpp-$l-$REPEAT_NUMBER -k $K -c $M -r $R"
}

# Produces the parameters to run DePermute
# Int -> String
function depermuteParams # ( k, repeatNumber )
{
    K=$1
    REPEAT_NUMBER=$2

    matrixFile    $K $REPEAT_NUMBER
    divCount      $K
    popSize       $K $DIVISION_COUNT
    eliteCount    $K $POP_SIZE
    mutationRate  $K
    commCount     $K $POP_SIZE

    DEPERMUTE_PARAMS="-i ${MAT_FILE}_ -e $ELITE_COUNT -r $R -p $POP_SIZE -c 2 -k
    $K -x $M -t 0.001 -m $MUTATION_RATE -s 100 -y 0 -n $COMM"
}

# Produces the path of the results file for CPU DePermute
# Int Int -> String
function cpuDePermuteResultsFile # ( k, repeatNumber )
{
    K=$1
    REPEAT_NUMBER=$2

    CPU_RESULTS_FILE=$DIR/Results-CPU-$K-$REPEAT_NUMBER.out
}

# Produces the path of the log file for CPU DePermute
# Int Int -> String
function cpuDePermuteLogFile # ( k, repeatNumber )
{
    K=$1
```

```
    REPEAT_NUMBER=$2

    CPU_LOG_FILE=$DIR/Log-CPU-$K-$REPEAT_NUMBER.out
}

# Produces the path of the results file for GPU DePermute
# Int Int -> String
function gpuDePermuteResultsFile # ( k, repeatNumber )
{
    K=$1
    REPEAT_NUMBER=$2

    GPU_RESULTS_FILE=$DIR/Results-GPU-$K-$REPEAT_NUMBER.out
}

# Produces the path of the log file for GPU DePermute
# Int Int -> String
function gpuDePermuteLogFile # ( k, repeatNumber )
{
    K=$1
    REPEAT_NUMBER=$2

    GPU_LOG_FILE=$DIR/Log-GPU-$K-$REPEAT_NUMBER.out
}

# Runs the experiment for a single value of K
# Int Int Int Int -> IO ()
function runExperimentAtK # ( k, procCount )
{
    K=$1
    PROC_COUNT=$2
    for ((L=0; L < $REPEAT; L++));
    do
        if (( $PROC_COUNT % $CONCURRENT == $PBS_ARRAYID ))
        then
            matrixFile $K $L
            if [ "$CLUMPP_UNDER_TEST" == "true" ]
            then
                # Run CLUMPP with the current problem
                clumppResultsFile $K $L
                clumppLogFile     $K $L
                clumppParams      $K $L
                echo "time ./CLUMPP $CLUMPP_PARAMS > $CLUMPP_RESULTS_FILE"
                echo "time ./CLUMPP $CLUMPP_PARAMS > $CLUMPP_RESULTS_FILE" >
    $CLUMPP_LOG_FILE
                (time echo '\r' | ./CLUMPP.exe $CLUMPP_PARAMS >
    $CLUMPP_LOG_FILE) 2> $CLUMPP_RESULTS_FILE
            fi
```

```
            # Run DePermute
193         depermuteParams $K $L
            if [ "$CPU_UNDER_TEST" == "true" ]
195         then
                cpuDePermuteResultsFile $K $L
197             cpuDePermuteLogFile     $K $L
                echo "time ./DePermute $DEPERMUTE_PARAMS > $CPU_RESULTS_FILE"
199             echo "time ./DePermute $DEPERMUTE_PARAMS > $CPU_RESULTS_FILE" >
    $CPU_LOG_FILE
                (time ./DePermute $DEPERMUTE_PARAMS > $CPU_LOG_FILE) 2>
    $CPU_RESULTS_FILE
201         fi
            if [ "$GPU_UNDER_TEST" == "true" ]
203         then
                gpuDePermuteResultsFile $K $L
205             gpuDePermuteLogFile     $K $L
                echo "time ./DePermute -g $DEPERMUTE_PARAMS > $GPU_RESULTS_FILE"
207             echo "time ./DePermute -g $DEPERMUTE_PARAMS > $GPU_RESULTS_FILE"
    > $GPU_LOG_FILE
                (time ./DePermute -g $DEPERMUTE_PARAMS > $GPU_LOG_FILE) 2>
    $GPU_RESULTS_FILE
209         fi
        fi
211 done
}
```

Listing C.12: Script to fall back to an environment outside of a Torque PBS environment for testing outside of the Wits cluster

```
# If we aren't running this in an array of nodes
2 if [ -z "$PBS_ARRAYID" ]
then
4     CONCURRENT=1
      PBS_ARRAYID=0
6     echo Running the test script outside of a torque pbs array...
else
8     CONCURRENT=$1
      shift 1
10 fi
```

Listing C.13: A script which is used to determine which platform the tests ought to be executed on.

```
GPU_UNDER_TEST="false"
2 CPU_UNDER_TEST="false"
CLUMPP_UNDER_TEST="false"
4 for param in $1 $2 $3
do
```

```
    if [ "$param" == "GPU" ]
    then
        GPU_UNDER_TEST="true"
    elif [ "$param" == "CPU" ]
    then
        CPU_UNDER_TEST="true"
    elif [ "$param" == "CLUMPP" ]
    then
        CLUMPP_UNDER_TEST="true"
    fi
done
```

C.3.1 Random Data Set Testing

Listing C.14: Script to run tests using random data

```
#!/bin/bash

# Experimental procedure for running CLUMPP, CPU-DePermute and GPU-DePermute on
    randomly generated data

# Special requirements:
# This script requires that several supporting files are present in  directory.
#  - RandomExperimentParameters.sh -- which contains parameters for the
    experiment
#  - PopCount.ini -- Population sizes for all sizes of K tested
#  - Mutations.ini -- Mutation rates for all sizes of K tested
#  - Elitism.ini -- Elitism rates for all sizes of K tested
#  - Comms.ini -- Proportion of the population which ought to be communicated
# If you can't find where a parameter is set it was probably set by
    RandomExperimentParameters.sh

# ===Experimental Setup===

source PBSArrayFallBack.sh
source ProgramUnderTest.sh
source RandomExperimentSetup.sh

# ===Experemental Procedure===

# Things I need to set for this experiment:
# Population size, mutation rate, divisions, elitist number, solutions
    communicated
# Should probably calculate these values from the convergance testing

# Construct a sub directory to generate the results in
mkdir -p $DIR
# Set the time format to real (i.e. seconds as a decimal number)
```

```
TIMEFORMAT=%R
PROC_COUNT=0
# Iterate through each value of K
for ((l=$K_START; l <= $K_END; l=$l + $K_STEP));
do
    runExperimentAtK $l $PROC_COUNT
    PROC_COUNT=$(($PROC_COUNT + 1))
done
```

Listing C.15: Script to run tests using random data to run on the GPU platform

```
#!/bin/bash
#PBS -N Rand
#PBS -l nodes=1:ppn=8,walltime=50:00:00,mem=50GB
#PBS -q WitsLong
#PBS -o /home/edward/Random.log
#PBS -t 0-14

if [ -z "$PBS_ARRAYID" ]; then
    bash TestRandom.sh 15 GPU
else
    cd /home/edward/Masters/masters-test-code/Test-Time
    bash TestRandom.sh 15 GPU
fi
```

Listing C.16: Script to run tests using random data to run on the CPU platform

```
#!/bin/bash
#PBS -N Rand
#PBS -l nodes=1:ppn=8,walltime=1:00:00,mem=50GB
#PBS -q WitsLong
#PBS -o /home/edward/Random.log
#PBS -t 0-14

cd /home/edward/Masters/masters-test-code/Test-Time
bash TestRandom.sh 15 CPU
```

Listing C.17: Script to run tests using random data to run using the CLUMPP program

```
#!/bin/bash
#PBS -N Rand-CLUMPP
#PBS -l nodes=1:ppn=8,walltime=100:00:00,mem=1GB
#PBS -q WitsLong
#PBS -o /home/edward/Random.log
#PBS -t 0-14

cd /home/edward/Masters/masters-test-code/Test-Time
bash TestRandom.sh 15 CLUMPP
```

C.3.2 Real Data Set Testing

Listing C.18: Script for experimental setup etc. when testing using real data sets.

```
#!/bin/bash

# The variables etc needed to run tests using CLUMPP and CPU and GPU - DePermute on Scott's provided
# data.

# NOTE: Re. how the data is arranged.
#  There are 50 directories and each contains a run for K: [2, 11]
#  This is the transpose of the way I usually do it.
#  What I could do is copy what I need to a local temp dir, run on that store the results delete the
#  temp dir and copy the next batch.

# ===Variables===

# Location of the data on the Wits cluster system
DATA_BASE_DIR=...

# Working dir for matrices and results to be written to
DIR=Real-Test
# The range of K values
K_START=2
K_END=11
K_STEP=1
# The number of R in each test
R=50
# The number of individuals in the study
M=909
# The number of times to repeat the experiment
REPEAT=2

# ===Functions===

# Produces the path to the base matrix file for the given K and repeat number
# Int Int -> String
function matrixFile # ( k )
{
    K=$1

    MAT_FILE=$DIR/Matrix-$K
}

source SharedSetup.sh
```

Listing C.19: Script to run tests using real data

```
#!/bin/bash

# Experimental procedure for running CLUMPP and CPU and GPU - DePermute on the "real" data sets
# which Scott has given me

# Special requirements:
# This script requires that several supporting files are present in  directory.
#  - RandomExperimentParameters.sh -- which contains parameters for the experiment
#  - PopCount.ini -- Population sizes for all sizes of K tested
#  - Mutations.ini -- Mutation rates for all sizes of K tested
#  - Elitism.ini -- Elitism rates for all sizes of K tested
#  - Comms.ini -- Proportion of the population which ought to be communicated
# If you can't find where a parameter is set it was probably set by RandomExperimentParameters.sh

# ===Experimental Setup===

source PBSArrayFallBack.sh
source ProgramUnderTest.sh
source RealDataExperimentSetup.sh

# ====Experimental Procedure===

# Construct a sub directory to generate the results in
mkdir -p $DIR
# Set the time format to real (i.e. seconds as a decimal number)
TIMEFORMAT=%R
PROC_COUNT=0
# Iterate through each value of K
for ((l=$K_START; l <= $K_END; l=$l + $K_STEP)); do
    if (( $PROC_COUNT % $CONCURRENT == $PBS_ARRAYID )); then
        matrixFile              $l
        #bash CopyRunToTemp.sh  $R $l          $MAT_FILE $DATA_BASE_DIR
        runExperimentAtK        $l $PROC_COUNT
        #bash DeleteMatrices.sh $R $K          $MAT_FILE
    fi
    PROC_COUNT=$(($PROC_COUNT + 1))
done
```

Listing C.20: Script to run tests using real data to run on the GPU platform

```
#!/bin/bash
#PBS -N Rand
#PBS -l nodes=1:ppn=8,walltime=50:00:00,mem=50GB
```

```
#PBS -q WitsLong
#PBS -o /home/edward/Random.log
#PBS -t 0-14

if [ -z "$PBS_ARRAYID" ]; then
    bash TestReal.sh 15 GPU
else
    cd /home/edward/Masters/masters-test-code/Test-Time
    bash TestReal.sh 15 GPU
fi
```

Listing C.21: Script to run tests using real data to run on the CPU platform

```
#!/bin/bash
#PBS -N Real
#PBS -l nodes=1:ppn=8,walltime=10:00:00,mem=50GB
#PBS -q WitsLong
#PBS -o /home/edward/Random.log
#PBS -t 0-8

if [ -z "$PBS_ARRAYID" ]; then
    bash TestReal.sh 15 CPU
else
    cd /home/edward/Masters/masters-test-code/Test-Time
    bash TestReal.sh 9 CPU
fi
```

Listing C.22: Script to run tests using real data to run using the CLUMPP program

```
#!/bin/bash
#PBS -N Real-CLUMPP
#PBS -l nodes=1:ppn=8,walltime=10:00:00,mem=1GB
#PBS -q WitsLong
#PBS -o /home/edward/Random.log
#PBS -t 0-8

cd /home/edward/Masters/masters-test-code/Test-Time
bash TestReal.sh 9 CLUMPP
```

Appendix D

Program Versions

Program	Version
	Home Desktop
Microsoft Windows	8.1 Pro
Microsoft Visual Studio	2013 Community Edition v12.0.31101.00
NVIDIA CUDA Toolkit	V7.0
Haskell Platform	2014.2.0.0
Git Bash Tools	1.9.5-preview20150319
Python	2.7.9
	AWS Machine
Red Hat Linux	Red Hat 4.8.2-16
GCC	(GCC) 4.8.2 20140120 (Red Hat 4.8.2-16)
Cuda Toolkit	6.5.14-1.12.amzn1
Bash	4.1.2(1)-release (x86_64-redhat-linux-gnu)
Python	2.6.9
	Wits Cluster Machines
Scientific Linux	6.3 (Carbon)
GCC	4.8.1
Bash	4.1.2(1)-release (x86_64-redhat-linux-gnu)
Python	2.6.6

Table D.1: Table of all programs used in the compilation and testing procedures

Appendix E

Test Platforms

Attribute	Value
System A	
CPU Model	AMD FX9370
CPU Core Clock	4.4GHz
L1 Cache Size	128KB
L2 Cache Size	256KB
L3 Cache Size	8MB
RAM	16GB DDR3 @1800Mhz
Graphics Card Model	NVIDIA Geforce GTX960
Graphics Card Core Clock	405 MHz
Graphics Card RAM Capacity	2048MB
System B	
Amazon Instance Model Id	g2.2xlarge
CPU Model	Intel Xeon E5-2670
Cache Size	20MB Smart Cache
RAM	15GB
Graphics Card Model	NVIDIA Grid K520
Graphics Card Core Clock	823-1063MHz
Graphics Card RAM Capacity	4GB
System C	
CPU Model	Intel E5-2620
Cache Size	15MB Smart Cache
RAM	32GB RAM

Table E.1: Table listing the specifications of the AWS and Home PC computers used during testing

Appendix F

Extra Details of Algorithm Implementations

Here the details of several algorithms and their implementations is discussed. In particular the implementation of parallel algorithms for the GPU are addressed.

F.1 Bitonic Sort

The bitonic sort algorithm is based on the observation that one can arrange numbers into a bitonic sequence (a sequence which increases in value up till its mid-point and then decreases) by creating sub-bitonic sequences for lengths increasing in powers of two. The creation of a sorted sequence from a bitonic sequence is trivial. A further observation is that the values can be swapped in parallel leading to a fast parallel implementation of the algorithm.

A natural way to understand the bitonic sort algorithm is by visualising the swaps which are made at each stage of the algorithm in a sorting network. *Figure F.1* illustrates the swaps made in an eight width network to sort a list of numbers [54]. Note that because of the lack of data dependencies in each box each swap may be performed in parallel.

In parallelising the bitonic sort, workers are stencilled across comparators in the bitonic sort network. This is achieved using the loops given in *Listing F.1* and the equation in *Equation F.1*.

Note that if j falls outside of the bounds of the sorting network then the corresponding thread breaks the inner loop. The complexity of the bitonic sorting algorithm is $O((lgn)^2)$ when there are half as many threads as there are values to sort. This in an improvement over the CPU version in itself, however it is limited by the number of threads when the population

Listing F.1: Loop descriptions for a bitonic sort

```
for ( l = CONCAT_SIZE; l > 1; l >>= 1 )
    for  ( L = offset; L < POP_SIZE; L += SORT_WORKERS )
        i = (L / (l / 2)) / l + L % (l / 2)
        q = i + (l / 2)
```

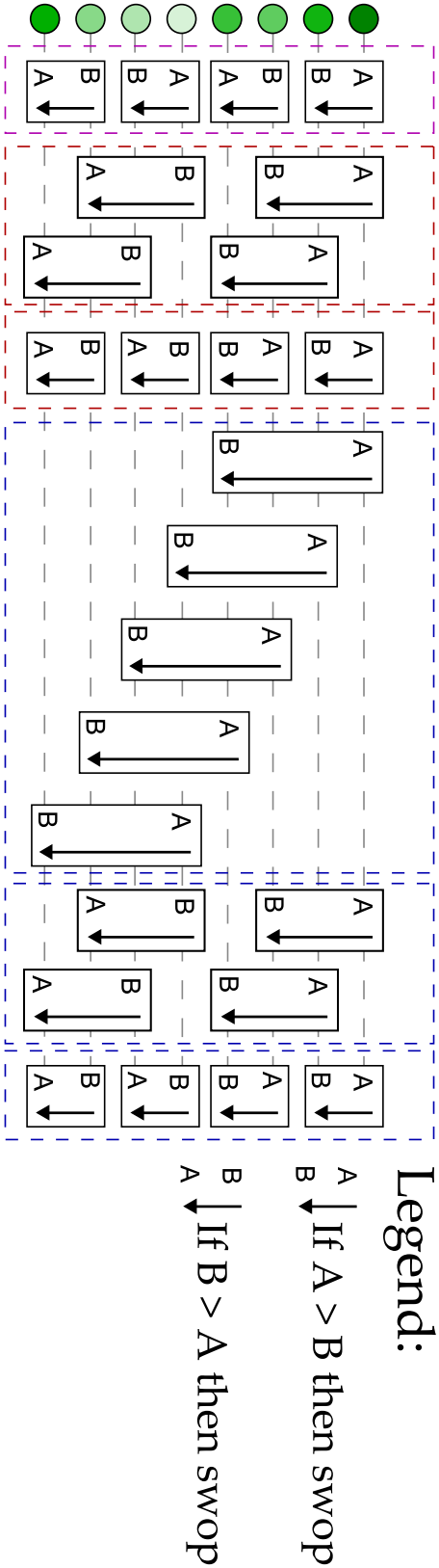


Figure F.1: Bitonic sort network for eight nodes

Listing F.2: Algorithm for the parallel prefix sum

```
prefixSums ( block )
    sums        <- []
    threadStart <- ( threadOffset + 1 ) * 2 - 1
    l           <- 0

    for l <- 0..length ( block )
        for L <- threadStart, threadStart + 2 * threadCount..length
            if ( L + 1 ) % l == 0
                sums[L] <- sums[L] + block[L - (l >> 1)]

    for l / 2, l / 4..1
        for L <- threadStart, threadStart + 2 * threadCount..length
            if ( L + 1 ) % / == 0
                sums[L + l / 2] = sums[l] + sums[L + l / 2]

sums[0] <- 1 / fitnesses[0]

for l <- 1, 1 + LOCAL_VALUES..POP_SIZE
    block <- copyBlock ( globalFitnesses[l:l + LOCAL_VALUES], LOCAL_VALUES )
    sums  <- prefixSums ( block )
    globalFitnesses[l:l + LOCAL_VALUES] <- sums
    sums[0] <- sums[LOCAL_VALUES - 1]
```

size grows.

F.2 Parallel Prefix Sum

The algorithm proceeds as given in *Listing F.2*. We now examine the parallel prefix sum in greater detail. The algorithm for generating prefix sums in parallel is two stage.

- Increasing partial sums are computed on the odd indices. The indices at increasing powers of two have the value, at the index of half their own index plus one, added to them.
- Increasing partial sums are computed on the even indices by a similar process.

There are often insufficient threads to compute every index in a block in parallel. Therefore each thread uses its index to compute a series of indices in the block where it should be updating the value. That thread alone will update the values at those indices, thus avoiding unnecessary memory fences.

Once prefix sums have been computed, a trivial stencil based loop is used to normalise all values in parallel using the value in the last index of the prefix sum buffer as the normalisation constant.

Bibliography

[1] D.H. Alexander, J. Novembre, and K. Lange. Fast model-based estimation of ancestry in unrelated individuals. *Genome Research*, 19:1655–1664, 2009.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52:56–67, 2009.

[3] P. Borovska, O. Nakov, and M. Lazarova. Parmetaopt – parallel metaheuristics framework for combinatorial optimization problems. In *IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems*. IEEE, 2009.

[4] Richard P. Brent. Some long-period random number generators using shifts and xors. *CoRR*, abs/1004.3115, 2010.

[5] T.M. Brewer. Instruction set innovations for the convey hc-1 computer. *IEEE Micro*, 30:70–79, 2010.

[6] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An idiom-finding tool for increasing productivity of accelerators. In *ICS '11: Proceedings of the international conference on Supercomputing*. Association for Computing Machinery, 2011.

[7] L. Luca Cavalli-Sforza. The human genome diversity project: past, present and future. *Nat Rev Genet*, 6(4):333–340, 04 2005.

[8] Yu-Sheng Chen, Antonel Olckers, Theodore G. Schurr, Andreas M. Kogelnik, Kirsi Huoponen, and Douglas C. Wallace. mtDNA Variation in the South African Kung and Khwe—and Their Genetic Relationships to Other African Populations. *The American Journal of Human Genetics*, 66(4):1362–1383, 2015/09/26.

[9] Convey Computer. *Convey Personality Development Kit Reference Manual*. Convey Computer, 2012.

[10] Convey Computer. *Convey Reference Manual*. Convey Computer, 2012.

[11] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series. Morgan Kaufmann, 2012.

[12] B.J. Copeland, C.J. Posy, and O. Shagrir. *Computability: Turing, Gödel, Church, and Beyond*. MIT Press, 2013.

[13] T. G. Crainic and M. Toulouse. Parallel strategies for meta-heuristics. In *Handbook on Meta-Heuristics*. Kluwer Academic Publisher, 2003.

[14] F. Dehne and K. Yogoratnam. Exploring the limits of GPUs with parallel graph algorithms. *CoRR*, abs/1002.4482:1–20, 2010.

[15] Audrey Delévacq, Pierre Delisle, and Michaël Krajecki. Parallel GPU implementation of iterated local search for the travelling salesman problem. In *Proceedings of the 6th international conference on Learning and Intelligent Optimization*, LION'12, pages 372–377, Berlin, Heidelberg, 2012. Springer-Verlag.

[16] K.A. Do, Z.S. Qin, and M. Vannucci. *Advances in Statistical Bioinformatics: Models and Integrative Inference for High-Throughput Data*. Cambridge University Press, 2013.

[17] Merrill M. Flood. The traveling-salesman problem. *Operations Research*, 4(1):61–75, 1956.

[18] G.H. Golub and C. F. Van Loan. *Matrix Computations*. JHU Press, 1996.

[19] I. Groat. *Digital Systems Design with FPGAs and CPLDs*. Newnes, 2008.

[20] P. Harish and P.J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. *HiPC'07 Proceedings of the 14th International Conference On High Performance Computing*, 14:197–208, 2007.

[21] Scott Hazelhurst. Truth in advertising : Reporting performance of computer programs, algorithms and the impact of architecture and systems environment. *South African Computer Journal*, 32:25–37, December 2012.

[22] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *J. Soc. Indust. Appl. Math.*, 10:1–15, 1962.

[23] S. Hong, Kim. S.K., T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 267–276, 2011.

[24] M. Jakobsson and N. A. Rosenburg. Clumpp: a cluster matching and permutation program for dealing with label switching and multimodality in analysis of population structure. *Bioinformatics*, 23:1801–1806, 2007.

[25] J. Jaros. Multi-GPU island-based genetic algorithm for solving the knapsack problem. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8, 2012.

[26] D. Joseph and W. Kinsner. Design of a parallel genetic algorithm for the internet. In *WESCANEX 97: Communications, Power and Computing. Conference Proceedings., IEEE*, pages 333–343, May 1997.

[27] I. Jung and C. Jeong. Parallel connected-component labeling algorithm for GPGPU applications. *Communications and Information Technologies (ISCIT), 2010 International Symposium*, pages 1149–1153, 2010.

[28] D Knuth. The art of computer programming. *Mathematics of Computation*, 1979.

[29] Pavel Krömer, Václav Snášel, Jan Platoš, and Ajith Abraham. Many-threaded implementation of differential evolution for the cuda platform. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1595–1602, New York, NY, USA, 2011. ACM.

[30] M.E. Lalami and D. EI-Baz. Gpu implementation of the branch and bound method for knapsack problems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1769–1777, 2012.

[31] Milena Lazarova and Plamenka Borovska. Comparison of parallel metaheuristics for solving the tsp. In *Proceedings of the 9th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, CompSysTech '08, pages 17:II.12–17:1, New York, NY, USA, 2008. ACM.

[32] C. Lin and L. Snyder. *Principles of Parallel Programming*. Pearson, Addison Wesly, 2009.

[33] Antonette M. Logar, Edward M. Corwin, and Thomas M. English. Implementation of massively parallel genetic algorithms on the maspar mp-1. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*, SAC '92, pages 1015–1020, New York, NY, USA, 1992. ACM.

[34] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Gpu-based island model for evolutionary algorithms. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1089–1096, New York, NY, USA, 2010. ACM.

[35] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(1):1–6, 2003.

[36] Nouredine Melab, Thé Van Luong, Karima Boufaras, and El-Ghazali Talbi. Paradiseo-mo-gpu: A framework for parallel gpu-based local search metaheuristics. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 1189–1196, New York, NY, USA, 2013. ACM.

[37] M.R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole. Modeling and predicting performance of high performance computing applications on hardware accelerators. *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, 26, 2012.

[38] B. Meyer, J. Schumacher, C. Plessl, and J. Forstner. Convey vector personalities - FPGA acceleration with an OpenMP-like programming effort? In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 189–196, 2012.

[39] Sayyed Ali Mirsoleimani, Ali Karami, and Farshad Khunjush. A parallel memetic algorithm on gpu to solve the task scheduling problem in heterogeneous environments. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 1181–1188, New York, NY, USA, 2013. ACM.

[40] M. Nordborg, T. T. Hu, Y. Ishino, J. Jhaveri, C. Toomajian, H Zheng, E. Bakker, P. Calabrese, J. Gladstone, R. Goyal, M. Jakobsson, S. Kim, Y. Morozov, B. Padhukasahasram, V. Plagnol, N. A. Rosenberg, C. Shah, J. D. Wall, J. Wang, K. Zhao, T. Kalbfleisch,

V. Schulz, M. Kreitman, and J. Bergelson. The pattern of polymorphism in arabidopsis thaliana. *PLOS Biology*, 2005.

[41] NVIDIA. *CUDA C Programming Guide*. NVIDIA, 2013.

[42] nVIDIA Coporation. Nvidia's next generation cuda compute architecture: Kepler tm gk110, 2012.

[43] Carina M. Schlebusch, Pontus Skoglund, Per Sjödin, Lucie M. Gattepaille, Dena Hernandez, Flora Jay, Sen Li, Michael De Jongh, Andrew Singleton, Michael G. B. Blum, Himla Soodyall, and Mattias Jakobsson. Genomic variation in seven khoe-san groups reveals adaptation and complex african history. *Science*, 338(6105):374–379, 2012.

[44] S.G. Shiva. *Advanced Computer Architectures*. Taylor & Francis, 2006.

[45] J. Silc, B. Robic, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond ; with 34 Tables*. Springer Berlin Heidelberg, 1999.

[46] Christopher C. Skiścim and Bruce L. Golden. Optimization by simulated annealing: A preliminary computational study for the tsp. In *Proceedings of the 15th Conference on Winter Simulation - Volume 2*, WSC '83, pages 523–535, Piscataway, NJ, USA, 1983. IEEE Press.

[47] A.B. Slomson. *Introduction to Combinatorics*. Discrete Mathematics and Its Applications. Taylor & Francis, 1997.

[48] S.I. Steinfadt. SWAMP+: Enhanced Smith-Waterman Search for Parallel Models. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 62–70, 2012.

[49] S. Suri. *Bioinformatics*. APH Publishing Corporation, 2006.

[50] David Uliana, Krzysztof Kepa, and Peter Athanas. FPGA-based HPC application design for non-experts. In *Rapid System Prototyping (RSP), 2013 International Symposium on*, pages 9–15, 2013.

[51] Maxwell Walton, Gary Grewal, and Gerarda Darlington. Parallel fpga-based implementation of scatter search. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, pages 1075–1082, New York, NY, USA, 2010. ACM.

[52] S. A. Williams. *Programming Models for Parallel Systems*. Wiley, 1990.

[53] Lai-Ping Wong, Rick Twee-Hee Ong, Wan-Ting Poh, Xuanyao Liu, Peng Chen, Ruoying Li, Kevin Koi-Yau Lam, Nisha Esakimuthu Pillai, Kar-Seng Sim, Haiyan Xu, Ngak-Leng Sim, Shu-Mei Teo, Jia-Nee Foo, Linda Wei-Lin Tan, Yenly Lim, Seok-Hwee Koo, Linda Seo-Hwee Gan, Ching-Yu Cheng, Sharon Wee, Eric Peng-Huat Yap, Pauline Crystal Ng, Wei-Yen Lim, Richie Soong, Markus Rene Wenk, Tin Aung, Tien-Yin Wong, Chiea-Chuen Khor, Peter Little, Kee-Seng Chia, and Yik-Ying Teo. Deep whole-genome sequencing of 100 southeast asian malays. *American Journal of Human Genetics*, 92(1):52–66, 01 2013.

[54] R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski. *Parallel Processing and Applied Mathematics, Part I: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009*. LNCS sublibrary. SL 1, Theoretical computer science and general issues. Springer, 2010.

[55] Weihang Zhu. A study of parallel evolution strategy: Pattern search on a gpu computing platform. In *Proceedings of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, GEC '09, pages 765–772, New York, NY, USA, 2009. ACM.