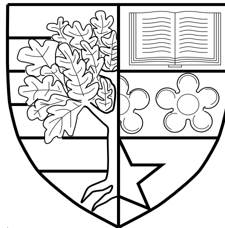
MULTI-ARCHITECTURE PARALLEL PROGRAMMING USING GPH, A FUNCTIONAL LANGUAGE

By

Mustafa KH. Aswad



Submitted for the Degree of

Master of Philosophy

at Heriot-Watt University

on Completion of Research in the

School of Mathematical and Computer Sciences

December 2002.

This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that the copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the prior written consent of the author or the university (as may be appropriate).

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Mustafa Kh. Aswad (Candidate)

(Supervisor)

(Date)

Contents

1 Introduction **1**

1.1 Overview 1

1.2 Contributions 2

1.3 Dissertation Outline 4

2 Background **7**

2.1 Parallel Computer Architectures 7

2.1.1 Beowulf Architecture 8

2.1.2 Sun SMP Architecture 9

2.2 Why Parallel Programming? 10

2.2.1 Parallel Program Development 11

2.2.2 Classification of Parallel Models 12

2.3 Architecture Independence 16

2.4 Architecture Independent Languages. 16

2.4.1 ZPL :A Machine Independent Programming Language for Parallel Computer 17

2.4.2 Parallaxis-III Architecture-Independent Data Parallel Processing 17

2.4.3 SAC Single Assignment C 18

2.4.4 CoPa 18
2.4.5 BSP Model 19
2.5 Architecture Independence Using Declarative Programming Languages. 19
2.6 Functional Programming 20
2.6.1 Theoretical roots and history of functional programming languages. 20
2.6.2 Functional Languages for Parallelism. 21
2.6.3 NESL 23
2.6.4 Eden 24
2.7 Haskell 24
2.7.1 GpH Parallel Functional Language 25
2.7.2 Parallelism in GpH. 25
2.7.3 Evaluation Strategies in GpH 26
2.8 GpH Compilers and Tools 27
2.8.1 The Hugs and GHCI Interpreter 27
2.8.2 The GHC Compiler and Sequential Runtime System. 28
2.8.3 GUM Parallel Runtime System 28
2.8.4 Time and Space Profilers 29
2.8.5 GranSim Simulator 30
2.8.6 Visualisation Tools 31

3 A Multiarchitecture Development Methodology 32

3.1 Overview 32
3.2 The Methodology Structure 33
3.2.1 Initial Sequential Stage 34
3.2.2 Sequential Optimised 34

3.2.3 Idealised Simulation Stage 34
3.2.4 Realistic Simulation Stage 35
3.2.5 Target Architecture 35
3.3 Enhancement of Architecture Independent Enhancement in GpH 36
3.4 Other Methodologies 36

4 Sequential Implementation 37

4.1 Problem Description 37
4.1.1 A Genetic Alignment Algorithm 37
4.2 Sequential Implementation 39
4.3 Alignment Example 40
4.4 Sequential Tuning 43
4.4.1 Development in Versions I, II and III 44
4.4.2 Development of Versions IV and V 47
4.4.3 Version VI using a Finite Map 48
4.4.4 Sequential Optimisation Discussion 50

5 Idealised Measurement 53

5.1 Introduction 53
5.2 Version I: Divide-and-Conquer 54
5.3 Version IIa: Parallelising Substring Sequences 55
5.4 Version IIb: Parallelising Form_pin 56
5.5 Version IIc: Parallelise Both Outer and Inner Loops 57
5.6 Version III: Clustering on Parallel Form_pin 57
5.7 Version IV: Parallelise all maps 58
5.8 Version V: Parallel all foldr 59

5.9 Idealised Optimisation Discussion 59

6 Two Simulated Architectures 64

6.1 Beowulf Simulation 64

6.2 Sun SMP Simulation 65

6.3 Discussion of Simulation Results 66

6.3.1 Idealised Simulation vs Realistic Simulation 66

6.3.2 Beowulf Simulation vs Sun SMP Simulation Comparison 67

7 Performance Measurements on Two Architectures 74

7.1 Real Measurement on Beowulf machine 75

7.2 Real Measurement on Sun SMP Machine 75

7.3 Discussion of Real Tuning 77

7.4 Critique of Multi-Architecture Methodology. 78

8 Enhancement of Architecture Independence in GpH 82

8.1 Overview 82

8.2 Extracting Architecture Characteristics 83

8.3 Generic Architecture Adapting Strategies 84

8.4 Architecture Adapting Strategies for Specific Application 88

8.5 Summary 90

9 Conclusions 94

9.1 Introduction 94

9.2 Achievements 95

9.2.1 Assessing a Multi-Architecture Parallel Programming Methodology 95

9.2.2 Extended The Architecture Independent Capabilities of GpH . . 96

9.3 Limitations 96

9.4 Future Work 97

A Source Code for The Genetic Alignment Program 98

A.1 Final Sequential Version 98

A.2 Complete Parallel Code of Version IIa 103

A.3 Finite Map Code 109

Bibliography 109

List of Tables

1 Sequential Profiling Summary 50

2 Idealised Simulation Input : 20 6 30. 61

3 Realistic 32-PEs Beowulf Simulation Input: 20 6 30 65

4 Realistic 32-PEs Sun SMP Simulation Input: 20 6 30 65

5 Real Beowulf Input: 20 40 30 on 4-processor 75

6 Real Beowulf Input: 20 60 30 on 30-processor 75

7 Summary Table of Real Measurement of SMP (20 40) on 4-processors 77

List of Figures

1 Distributed Memory and Shared Memory MIMD Architectures 8

2 CoPa Complexity Preserving Compilation 19

3 Basic Coordination Constructs in GpH 26

4 The parList & parMap Parallel Strategies 26

5 The Multi-Architecture Program Development Model 33

6 Input Sequences and the Aligned Output Sequences 38

7 Final Alignment Figure. 38

8 Functions Call Chart for Versions I,II,III. 43

9 The Align Chunk Function Sequential Code 44

10 The Bestpin Function 44

11 Divide and Conquer Sequential Code and Diagram. 45

12 Heap Profile of Initial Version. 46

13 Old and New Code of Extract_max_pin Function. 46

14 Old and New Code of Longest_pin Function. 47

15 Heap Profile of the Final Sequential Version. 48

16 The Modified Form_pin function of Version IV & V. 49

17 The Modified Functions to implement the Finite Map (Version VI). . . . 52

18 The Strategies Required for Parallel Divide Function 53

19 Idealised Simulated Profile of Version I 54

20 A Partial from Time Profile of the Final Sequential Version. 55

21 Divide and Conquer Process Diagram for Divide Function 56

22 Sequential and Parallel Code of Substring Function. 57

23 The Idealised Activity Profile of Substring Sequences Function (IIa). . 58

24 Sequential and Parallel Code of From_pin Function (Inner Loop). 59

25 The Idealised Activity Profile of From pin Function (IIb) 60

26 Cluster Function and Modified Substring Function. 61

27 The Idealised Activity Profile for Clustering version (III). 62

28 The Idealised Activity Profile for parMap version IV 62

29 New ParfoldList and Extractmaxpins Function. 63

30 The Idealised Activity Profile for Version V Input 20 6 30 63

31 The Activity Profile for Idealised vs Simulated Beowulf(Version I, IIa, and IIb) 68

32 The Activity Profile for Idealised vs Simulated Beowulf (Version III, IV, and V) 69

33 Activity Profile of Beowulf and Sun for Version IIa 70

34 Speedup vs Numbers Of PES (Simulated Beowulf & SMP) 72

35 Chunk Size vs Speed up for Both Beowulf and SMP Architectures . . . 73

36 Speedup vs Numbers of PEs Real (Beowulf & SMP) 76

37 Actual profiles of version IIa of Real Beowulf & SMP, Input 20 60 . . . 81

38 A New GpH Structure 83

39 A Number of PEs Function 84

40 The New Functions already Built in Architecture Model 85

41 The New General Divide Conquer Function 86

42 The New parMapPe Relative Speedup for Beowulf 87

43 The New parMapPe Relative speedup for SunSMP (Input 20 40) 88

44 A New Divide Function 89

45 Divide Function Diagram when it called by 2 PEs 89

46 Activity Profile for New and Old Divide Function (20-Processors) 92

47 The New Divide Function Relative speedup for Beowulf 93

48 The New Divide Function Relative speedup for Sun SMP 93

Acknowledgements

My praises to God for giving me the good health, the strength of determination and support to finish my work successfully. I am grateful to Phil Trinder and Hans Wolfgang Loidl, my supervisors. They were always ready to listen to my work with patience. They pulled me through the different attempted research area with very helpful suggestions and with a clear sense of direction. I also wish to thank my company (Azzawyia Oil Refinery) for their great efforts in giving financial support to enable me to complete my MPhil study. Finally, I wish to thank my family for their love and affection.

Abstract

This thesis investigates the use of a high level functional language GpH (Glasgow parallel Haskell) for architecture independent parallel programming. The aim is to provide acceptable performance across a wide range of parallel architectures with minimal programming effort. High level languages are a good alternative for architecture independent parallelism as they are designed to hide most architecture-dependent details from the programmer.

The thesis describes the first systemic investigation of a newly-proposed multi-architecture programming methodology for GpH. The methodology has two main phases: an architecture independent phase of idealised parallelisation, and an architecture dependent phase of accurate performance prediction and tuning. The methodology is used to develop a substantial application for two architectures with different hardware characteristics: a Beowulf cluster and Sun SMP. Sequential tuning improves performance from 224s to 19s, substantially owing to the elimination of intermediate data structures. Seven alternative parallel versions of the program are developed and evaluated using a simulated idealised architecture. Realistic simulation of the two target architectures accurately predicts the version that delivers the best performance in practice. Ultimately acceptable speedups are achieved on both architectures: 7.5 on a 30-processor Beowulf and 1.8 on a 4-processor Sun SMP. Transfer between architectures does not require source code changes.

To improve the architecture independence of GpH new parallel coordination constructs for GpH have been designed, implemented and measured. The primitives extract key architecture specific properties of the machine and use them to control coordination, often without exposing the properties to the programmer. Improved parallel performance is demonstrated using the primitives.

Chapter 1

Introduction

1.1 Overview

The development of a parallel program presents a set of problems that do not arise in the development of sequential software. Principal among these problems is the influence of the target architecture on the program development process. In particular the performance tuning process is very sensitive to the target parallel architecture. Consequently, the development of a parallel program is typically carried out in an architecture-dependent manner, with a fixed target architecture [1]. Traditional approaches of parallel programming explicitly specify most parallel aspects such as communication, task synchronisation, and work distribution. An alternative approach is to hide most of these aspects behind a high level language implementation. A high level language enables flexible programs and more portability with an acceptable performance across a wide range of parallel architectures [2]. However, high level programming models are still less efficient compared with low level languages.

The goal of architecture independent parallel programming languages is that *the programs can be transferred from architecture to architecture without sacrificing much efficiency or requiring significant redevelopment* [3]. High level languages are potentially architecture independent as parallel coordination is specified at a high level of abstraction, i.e. without reference to a specific underlying machine. A parallel coordination describes how the computation are arranged on the virtual machine, including aspects such as thread creation, placement and synchronisation. The challenges are to produce effective and efficient implementations of the high-level coordination, and to develop methodologies to develop software systematically for multiple architectures.

Glasgow parallel Haskell (GpH) is a functional language with a high level parallel programming model designed to deliver good performance across a number of parallel architectures. It is implemented using the Glasgow Haskell Compiler (GHC), with a parallel runtime system (GUM), that dynamically manages many of the aspects of parallel execution and automatically adapts its behaviour to the underlying architectures [4].

This thesis investigates a proposed multi-architecture methodology for developing GpH parallel programs and extends this methodology [5]. The methodology was used to develop multi-architecture parallel program for the first time.

1.2 Contributions

The main contribution of this thesis is to assess architecture independence of high level parallel functional languages, particularly GpH. More specifically, the contributions are as follows:

The first systematic evaluation of a multi-architecture development methodology. The methodology has two main phases: an architecture independent phase of idealised parallelisation, and an architecture dependent phase of accurate performance prediction and tuning. In the development of parallel programs most of the work is done in the architecture independent phase. Sequential optimisation is independent of parallelisation and delivers a good sequential program before inserting any parallelism. The sequential optimisation required to detect the space leak problem which is a common problem in the non-strict functional language. The idealised simulation enables the programmer to simulate the program on different parallel machines including the idealised machine with an infinite number of processors and zero communication costs. The GranSim simulator [6] provides considerable flexibility to emulate different architectures including the idealised machine which gives a good indicator of the maximum parallelism that can be obtained. If only a small amount of parallelism is obtained on the idealised simulation then very little is possible on any architecture. In the architecture dependent phase the simulator is parameterised to emulate the target machine. The final stage is to execute the parallel program on a real machine using the GUM runtime system provided by GpH.

In this thesis, the methodology is used to develop a parallel program for genetic alignment targeting two parallel architectures with different hardware characteristics: a distributed memory Beowulf cluster and a shared memory Sun SMP. This thesis then assesses the performance of the resulting programs and the architecture independence of the program development process.

Extending the architecture independent capabilities of GpH. To improve the architecture independence of GpH proposes new parallel coordination constructs are

proposed. The primitives extract key architecture specific properties of the machine and use them to control coordination, often without exposing the properties to the programmer. In particular, refinements of data-parallel and divide-and-conquer coordination are presented. The thesis discusses the importance of the architecture specifics extracted and extends the programming methodology with a new module exploiting this information.

In addition to the main contributions above this thesis surveys a number of architecture independent parallel programming languages and discusses how each language achieves the goal of architecture independent parallelism.

1.3 Dissertation Outline

Chapter 2 presents an overview of various approaches towards architecture independent parallel programming models. In addition the two different parallel architectures used in this investigation are described.

Chapter 3 gives a detailed description of the proposed multi-architecture programming methodology. It highlights the tools used in the methodology and the importance of each stage.

Chapter 4 describes the genetic alignment program and its sequential implementation and performance tuning. The sequential optimisation is an important stage of the multi-architecture development methodology. To achieve a good parallel performance, it is necessary to start with a good sequential version.

Chapter 5 describes the idealised parallelisation of the genetic alignment program, identifying seven different sources of parallelism.

The different parallel versions are tested using the GranSim simulator parameterised to emulate an idealised machine. The idealised machine has zero communication costs, and an unlimited number of processors. The primary goal of this stage is that the program exposes the maximal amount of parallelism that can be achieved from the algorithm.

Chapter 6 describes the measurement of the different parallel versions of the genetic alignment program on two simulated parallel architectures. In this stage, the GranSim simulator is parameterised with the key parameters of the target architectures.

Chapter 7 describes the measurement of the parallel versions of the program on the two architectures and summarises the difference between simulation and real measurement. The programs are executed on the architectures using the GUM runtime system provided by GpH. Usually minimum code changes are required in this stage.

Chapter 8 describes the improvement of the architecture independence of GpH, which employs the underlying architecture for controlling the parallelism. It describes how the key architecture-specific property of processors number is extracted. This property is used at two levels: the strategic level, where the new parameters are hidden from the programmer, and the application level where it can be used explicitly in order to refine coordination. Also it describes the new parMapPe and divide functions which employ the extracted property.

Chapter 9 contains a summary of the achievements and limitations of the work presented in thesis. It evaluates the importance of the development methodology and its environment tools.

The appendices contain versions of the genetic alignment program and are organised as follows: Appendix A.1 contains the code for the optimised sequential version; Appendix A.2 contains the code for the best parallel version (IIa) which delivers best speedup on both architectures. Appendix A.3 contains modified functions for the `finite-Map` implementation.

Chapter 2

Background

This chapter describes some parallel architecture and parallel programming issues. First, a brief description of the construction of parallel platforms is given, covering the Beowulf cluster architecture and Sun SMP architecture. Both machines are targeted for use in this investigation project. Second, parallel programming development and the types of parallel programming models are discussed.

2.1 Parallel Computer Architectures

Parallel programming will be useful if a general parallel machine (such as a von Neumann sequential machine model) can be defined. This machine model must be simple to program and the programs developed for the model executed with reasonable efficiency on real computers [7].

Parallel computers consist of multiple processors, memory modules, and an interconnection network. The distinction between the parallel computer architectures is determined by the arrangement of the above components. The processors used in parallel computers are increasingly exactly the same as processors used in single-processor

systems [1].

The Multicomputer consists of a number of von Neumann computers linked by an interconnection network. Each computer performs its own program. This program may access local memory and may send and receive messages over the network. Messages are used to communicate with other computers or, equivalently, to read and write remote memories. This model fits the parallel programming requirement.

The SIMD Machines are array processors. They typically consist of large collection of small processing elements. All processors execute the same program on a different piece of data. MIMD machines consist of a number of processors which execute a separate stream of instructions on their own data.

The MIMD machine may be distributed memory or shared memory. Distributed memory means that memory is distributed among the processors, rather than placed in a central location. In shared memory all processors have shared access to a common memory, often via a bus. Figure 1 shows the distributed-memory and shared-memory MIMD machines.

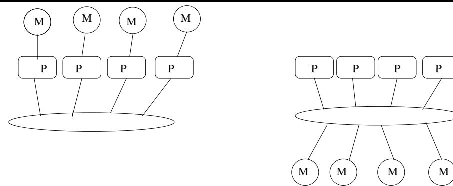


Figure 1: Distributed Memory and Shared Memory MIMD Architectures

2.1.1 Beowulf Architecture

A Beowulf cluster is a distributed-memory (MIMD) multicomputer architecture used for parallel computations [8]. It may contain a server node, which acts as a gateway to

the outside world, assigns IP addresses etc. It is built from stock hardware, therefore it is cheap to build. Beowulf also uses commodity software such as the Linux operating system, Parallel Virtual Machine (PVM) [9] and Message Passing Interface (MPI) [10]. Large Beowulf machines might have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases client nodes in a Beowulf system are dumb, and the dumber the better. Nodes are configured and controlled by the server node, and do only what they are told to do. In a disk-less client configuration, client nodes do not even know their IP address or name until the server tells them what it is. One of the main differences between Beowulf and a Cluster of Workstations is the fact that Beowulf behaves more like a single machine rather than many workstations. In most cases client nodes do not have keyboards or monitors, and are accessed only via remote login or possibly serial terminal. Beowulf nodes can be thought of as a CPU + memory package which can be plugged in to the cluster, just like a CPU or memory module can be plugged into a motherboard [8].

Heriot Watt University had already acquired a Beowulf cluster which will be used for the project experiment. It is a 32-node Beowulf cluster consisting of Linux Red Hat 6.2 workstations with a 533 MHz Celeron processor, 128 Kb cache, 128 Mb of DRAM, 5.7 Gb of IDE disk, connected through a 100Mb/s fast Ethernet switch with latency 142 microseconds.

2.1.2 Sun SMP Architecture

Shared memory is a model for interactions between processors within a parallel system. Systems like the multi-processor Pentium machines running Linux physically share a single memory among their processors, so that a value written to shared memory by

one processor can be directly accessed by any processor. Alternatively, logically shared memory can be implemented for systems in which each processor has its own memory by converting each non-local memory reference into an appropriate inter-processor communication. Physically shared memory can have both high bandwidth and low latency, but only when multiple processors do not try to access the bus simultaneously [11]. The Sun SMP which will be used for this project consists of 4-processor with clock speed of 250 MHz, and it has latency under the PVM layer between nodes as 109 microseconds.

2.2 Why Parallel Programming?

Parallelism is a very interesting perspective in understanding computer architectures because it applies at all levels of design, and interacts with essentially all other architectural concepts. A parallel computer is a collection of processing elements that communicate and cooperate to solve large problems quickly. The most important aspects in the design of a parallel computer are: the number of processors, the processing power of each processor, communication and cooperation between the processors, the way of transferring data, the interconnection manner used and the operations available to sequence the actions carried out on different processors, the primitive abstractions that the hardware and software provide to the programmer, and finally translation of all to performance [12]. These issues are reflected in low-level parallel languages. In contrast, to this hardware view, Skillicorn [1] summarises the demand for parallel programming as follows:-

- The real world is inherently parallel, so it is natural and straightforward to express computations about the real world in a parallel way, or at least in a way that does not preclude parallelism.

- Parallelism makes available more computational performance than is available in any single processor, although getting this performance from parallel computers is not straightforward.
- There are limits to sequential computing performance that arise from fundamental physical limits such as the speed of light.
- Even if single-processor speed improvements continue on their recent historical trend. But the costs of designing and fabricating each new generation of uniprocessors are unlikely to drop.

2.2.1 Parallel Program Development

The most important issues in writing a parallel program are: partitioning a program into tasks, mapping tasks onto a processor, and arranging for tasks to communicate safely [13]. The above issues make the parallel programming quite difficult. Most of the current research studies aim to produce a parallel programming language which can make the parallel programming easier. In this section some of parallel programming models which have similar objectives to the GpH model are surveyed. A brief description of how each model is formed is given.

First, the high level language makes the programmer's task become easier because there is no longer need for making accurate judgement and decisions about parallelism. Consequently, the development and maintenance of programs become easier, and there is less scope for programmer error. Second, programs become portable, because there are no detailed low level descriptions of parallelism inserted to the program. In other word the program does not contain low level descriptions for a particular platform.

Most of the current research studies aim to produce a parallel model which separates

the high-level properties from low-level ones. A model should be an abstract machine providing certain operations to the programming level above, and the requirement of implementing these operations on all architecture. In other words, for a parallel model to be useful, it must address both issues, abstraction and effectiveness. The development of a parallel program must address the following issues, according to [14]:

- The parallel program should specify the useful Parallelism from the problem description. The algorithm must be able to determine the potential parallelism inherent in the problem. This involves the splitting the program into sequential chunks that can be executed in parallel. However, the algorithm must be aware of the cost of communication between processors.
- In mapping, the generated tasks of the program must be mapped down to the physical resources of the target architecture. This may involve grouping tasks together and scheduling their execution on the same processor.
- Managing process interaction is not just a matter of writing a number of sequential threads of code. These threads will normally have to cooperate in some way.
- When ensuring program correctness, as parallel programming is more complex than sequential programming, there are more things than can potentially cause errors. While verifying the correctness and proper working of sequential software is demanding enough, doing the same for parallel software is much harder.

2.2.2 Classification of Parallel Models

The classification is based on how different models control parallelism. A brief description of some of the current models will be given, along with different ideas presented

by the existing models showing how they achieve parallelism. Parallel programming languages has been classified by Skillicorn [1] into six categories:

1. Models that abstract from parallelism completely. Such models are fully implicit and describe only the purpose of a program and not how it is to achieve this purpose.

2. Models in which parallelism is made explicit, but decomposition of programs into threads, mapping, communication, and synchronisation are made implicit.

3. Models in which parallelism and decomposition are explicit, but mapping, communication, and synchronisation are implicit.

4. Models in which parallelism, decomposition, mapping are explicit, but communication , and synchronisation are implicit.

5. Models in which parallelism, decomposition, mapping, communication are explicit, but synchronisation is implicit.

6. Models in which every thing is explicit.

Implicit Parallelism.

Implicit parallelism (1) is automatically exploited by the compiler and the run-time support system [15]. The programmer does not have to specify parallelism explicitly using special language constructs, compiler directives, or library function calls. The underlying system is hidden from the user, which shields programmers from the increased complexity of parallelism and shifts the burden to the compiler writer [13]. The most popular approach of implicit parallelism is automatic parallelisation of sequential programs. The advantages of the implicit parallelism approach are that existing sequential

software can be reused for parallel computers. Programmers familiar with sequential languages do not need to know about parallel programming or parallel architectures to exploit their parallelism. In addition, it is easier to understand the semantics of implicit programs than of explicit ones. Id [16] is one example of implicit parallel programming languages.It designed by members of the Computation Structures Group in MIT's Laboratory for Computer Science, and is used for programming dataflow and other parallel machines. Id programs are implicitly parallel to a very fine grain.

Semi-Implicit Parallelism

In semi-implicit parallelism (2-5), the programmer is required to insert annotations into a program to tell the compiler where a potential parallelism is useful. These annotations are used to control the parallel behaviour of the program but they hide in their implementation all low level details. An example of this approach is GpH. GpH uses a `par` and `seq` combinator to exploit parallelism in the program [17]. The approach exploits data parallelism by performing a high order function on all elements of a large data structure at the same time [18].

Skeleton Parallelism

Cole [19] has proposed to use skeleton algorithmic as a technique to parallelise functional languages and program parallel machines. The idea is capture common patterns of parallel computation in Higher Order Functions (HOFs). The common parallel coordination is hidden from the programmer, only HOFs are used to introduce parallelism. The major advantage of skeletons is the portability of parallel programs written using this approach. This results from the separation of meaning behaviour for each skeleton.

Coordination Languages

Caliban is one of the most known functional coordination language introduced by Paul Kelly. The Caliban coordination language provide controls to statically map the parallel tasks to the processors. Caliban is an annotation mechanism which specifies how a Haskell program is executed in parallel [20]. Other approaches such as Linda introduce a completely new coordination language layer that controls the dynamic execution of sequential program fragments written in a convebtional programming language.

Explicit Parallelism.

In explicit parallelism (6), the programmer informs the compiler where parallel evaluation should take place. Here the main responsibility of explicit parallel programming is put back onto the programmer, whose skill and knowledge is instrumental to the efficiency of the parallel implementation. High-level data-parallel languages such as High Performance Fortran (HPF) [21] and Fortran D [22], offer a simple and portable programming model for parallel, scientific programs. In such languages, programmers specify parallelism abstractly using data layout directives, and a compiler uses these directives as the basis for synthesising a program with explicit parallelism and inter processor communication and synchronisation. In Hudak's para-functional programming the programmer can schedule expressions to be evaluated sequentially or in parallel, and even specify on which processor a given expression should be evaluated.

2.3 Architecture Independence

Two approaches to introducing explicit parallelism are by using libraries or by adding language extensions. One of the architecture independent approaches is a parallel program which uses the standard procedure libraries such as MPI, PVM, and OpenMP [23]. These libraries are widely used and run on almost all parallel platforms, because they are supported by machine vendors. The processes that compose a parallel application can run on different machines as part of the same program. To use these libraries the programmer must program all of the process decomposition, placement, and communications explicitly. Another approach is by using the high level languages, some examples of these are High Performance Fortran (HPF), parallel C++, Java, and Declarative languages, such as Glasgow parallel Haskell(GpH) [1].

2.4 Architecture Independent Languages.

The popular class is data parallel language. It is oriented much more toward the machine than to the human programmer. These languages were simply abstractions of the von Neumann organisation of the machines on which they were implemented. In contrast, declarative programming languages are claimed to be particularly human oriented [24]. The characteristic of data parallel programming models is that the operation can be performed in parallel on each element of a large regular data structure, such as list or array. The program is a logically single thread of control, carrying out a sequence of either sequential or parallel steps [25].

2.4.1 ZPL :A Machine Independent Programming Language for Parallel Computer

ZPL [3] is one of the imperative data parallel languages. It provides high level semantics that explicitly represent parallel operations. The ZPL compiler uses Ironman machine independent communication interface to provide a separation of concerns. The compiler determines what data to send and when it can legally be sent. Machine specific libraries then specify how to send the data, which allows each machine to use the low level mechanism that is most suitable. In order to obtain efficient parallel performance on parallel computer, ZPL achieves a good performance by executing the program on sequential computer similar to the GpH model.

2.4.2 Parallaxis-III Architecture-Independent Data Parallel Processing

Parallaxis-III [26] is imperative parallel language based on modula-2, extended by data parallel concepts. The language is fully machine independent across data parallel architectures; as a result a program written in Parallaxis runs on different parallel computer systems. There is a Parallaxis simulation system with source level, debugging and tools for visualisation and timing. Parallaxis compilers can be used to generate parallel codes for data parallel systems. The simulation environment allows both the study of data parallel fundamentals on simple computer systems and the development of parallel programs, which can later be executed on expensive parallel computer systems. The central point of Parallaxis is programming on a level of abstraction with virtual PEs (processor elements and virtual connections). Moreover, in the algorithmic description, every program includes a connection declaration in functional form. This means that the desired connection topology is specified in advance for each program and can be

addressed in the algorithmic section with symbolic names instead of complicated arithmetic index or pointer expressions. However, full-dynamic data exchange operations are also possible. Parallaxis provides two compilers (seq.p3 and par.p3) to allow a programmer to examine her/his program on sequential and parallel architecture.

2.4.3 SAC Single Assignment C

SAC [27] is a strict first-order functional language with implicit parallelism and implicit thread interaction, optomised for array processing. Array operations in Sac are based on elementwise specifications using so-called With-loops. These language constructs are also well-suited for concurrent execution on multiprocessor systems.

2.4.4 CoPa

CoPa is a high-level language for processing nested sets, bags, and sequences (a generalisation of arrays and lists). CoPa includes most features found in query languages for object-oriented or object-relational databases, and has, in addition, a powerful form of recursion not found in query languages. CoPa has a formal declarative definition of parallel complexity, as part of its specification [28]. CoPa achieves architecture independence by using a parallel vector machine model (BVRAM) which supports the complexity-preserving compilation of CoPa's high-level constructs and efficient implementation on a variety of architecture. The language provides a logP simulator to measure the parallel aspects, such as communication cost.

CoPa achieves the architecture independence with a way similar to that in GpH; Figure 2 shows a parallel compilation technique for CoPa. The BVRAM (Bounded Vector Access Machine) provides support to the high level construct and efficient implementability on different architectures.

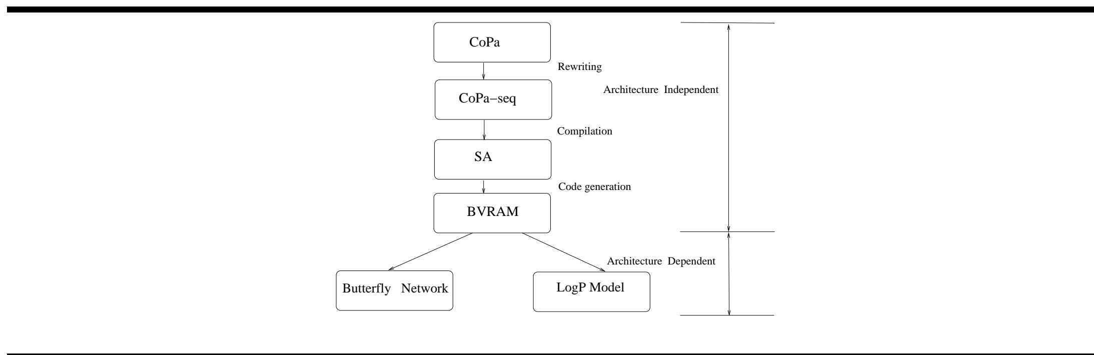


Figure 2: CoPa Complexity Preserving Compilation

2.4.5 BSP Model

The model uncouples the two fundamental aspects of parallel computation communication and synchronisation. This uncoupling is the key to achieving universal applicability across the whole range of parallel architectures. BSP programs are written in super steps which are global operations of the entire machine. Each super-step consists of three sequential phases: (1) a computation phase in which each processor computes with locally-held values, (2) a communication phase in which communication between processing elements takes place, and (3) a barrier synchronisation. Transferred data is not visible to the programmer code at its destination until after the barrier synchronisation ends the super step in which it was transferred [29].

2.5 Architecture Independence Using Declarative Programming Languages.

Declarative Programming languages, include the logic and functional languages. They are characterised by a very high level of abstraction. This allows the programmer to focus on what the problem is and offers many clear details of how the problem should be solved. Declarative languages have opened new doors to automatic exploitation

of parallelism. Their focusing on a high level description of the problem and their mathematical nature have turned into positive properties for implicit exploitation of parallelism [30].

In logic programming, all parallelisation is performed by the compiler, without any details being supplied by the programmer. The possibility of shared variables in different sub goals makes parallelisation more difficult. Implicit parallelism can be subdivided into AND and OR parallelism. OR-parallelism involves matching a single goal to many clauses simultaneously. AND-parallelism involves the simultaneous resolution of several goals in a clause [24]. This thesis does not discuss the logic programming further.

2.6 Functional Programming

In this section the advantages of functional programming and how parallelism is achieved in functional programming, in particular the GpH Glasgow parallel Haskell model, will be discussed.

2.6.1 Theoretical roots and history of functional programming languages.

This section is based on Peyton Jones, Plasmeijer ,and Lisper [31, 32, 33]. Functional programming is based on the Lambda Calculus a branch of logic, developed in the 20's and 30's. The Lambda Calculus is a simple formal language of functions, and the first developments were by Schonfinkel (1924), and Curry (1930): they defined a variation called combinatory logic. Church(1932/1933) [34] then defined the first version of the actual Lambda Calculus. These early logicians had no intention to define a programming language.

The first functional language has defined by McCarthy defined around 1960 [35]. In the late 1970's, Backus defined the FP language [36]. The important idea in FP is the standard set of higher order functions which take functions as arguments or return them as results. Around the same time, researchers at University of Edinburgh defined the ML language ("Meta-Language"), with polymorphic type inference and a sophisticated module system.

In the early/mid 1980s a number of lazy functional languages were developed, such as MIRANDA [37] and LML [38]. Lazy or non-strict languages try to defer evaluation of expressions until the result is needed. Haskell [39] was defined in 1990 as the standard, non-strict, higher order functional language. It contains many of the features from earlier functional languages, such as higher order functions, type inference, and non-strict semantics.

2.6.2 Functional Languages for Parallelism.

Functional programming style is similar to the data flow model inasmuch as programs are built of blocks that transform input to output without side-effects. These blocks are called functions and originate from the mathematical idea of a function. The properties of pure functions ensure that rewriting does not influence the result of computations. Therefore, automatic optimisation parallelisation and transformation for optimisation on computation are possible and parallelisation are possible.

Functional languages are general purpose, high level programming languages supporting programming at a higher level of abstraction than conventional imperative languages like FORTRAN and C. Programming in functional languages is a declarative activity which involves specifying only what is to be computed, while imperative programming specifies also the order of the computation steps.

Junaidu [40] says that a major distinction between modern functional languages and their imperative language with the same properties is that the former do not allow assignments (i.e., destructive updates) to memory locations. Alternatively, functional languages use only declarations (which are technically different from single assignments) whereby a variables value in a program, once declared, does not change. The lack of assignments facilitates higher level programming since the concern of programming is separated from that of low level housekeeping of recycling memory locations enforced by repeated assignments. The absence of assignments in functional languages serves as an important prerequisite which confers on these languages a useful mathematical property. This property ensures that since there are no side-effects, the value of an expression in a program depends only on the values of its syntactically correct constituent expressions and not, for example, on the order in which the expressions are evaluated.

Functional languages are often classified according to their semantics, into strict, non-strict and lenient. Eager evaluation is usually used to implement strict semantics while lazy evaluation is the implementation technique often used to implement non-strict semantics. The Third (Lenient) evaluation combines non-strictness with strict evaluation.

A function is strict if it depends on its argument. A non-strict function is a partial function that may be defined even when one of its arguments is not defined. Lazy evaluation starts evaluating the function body, evaluating the function's arguments only as and when they are used. Lenient evaluation starts the evaluation of the function in parallel with the evaluation of all the arguments of the function, and it supports functions which return results even when their computation may not terminate.

Functional languages provide higher order functions. One common example of higher order functions is a function which maps another function over a list. In principle

each list element may be processed in a separate processor. So the programmer need only provide new argument functions to introduce parallelism [41].

2.6.3 NESL

NESL is one of the most successful parallel functional languages. It is a strict, strongly-typed, data-parallel language with implicit parallelism and implicit thread interaction. It has been implemented on a range of parallel architectures, including several vector computers. NESL fully supports nested sequences and nested parallelism, and has the ability to take a parallel function and apply it over multiple operations over the data. NESL is loosely based on ML functional language. The most important parallel feature in NESL is the apply-to-each construct. This construct uses a set-like notation. NESL also provides a performance model for calculating the asymptotic performance of a program on various parallel machine models. This is useful for estimating running times of algorithms on actual machines. The NESL compiler compiled the NESL code to an intermediate vector code (VCODE) format. The vector instructions in this language-independent VCODE format are then mapped to a library of low-level, architecture specific. NESL uses a method based on asynchronous processor groups to reduce communication and a run-time load-balancing system to cope with dynamic data distributions. This is done by translating the user's algorithm into ANSI C with MPI calls, and linking this code with an MPI (Message Passing Interface) library [42]. By using the performance model provided the programmer can tune his application to achieve better performance on different architectures.

2.6.4 Eden

Eden [43] coordinates parallel computations using explicit process creation and interconnection, enabling the programmer to define arbitrary process networks. Thread interaction can be either implicit, via shared variables and function parameters on process creation time, or explicit via communicating parameters to processes during the process life time. The language uses a closed system model with location independence. The programmer typically starts with a specific process network in mind and models this network using explicit processes. Evaluation strategies may also be required. Eden offers more possibilities for tuning the parallel performance.

2.7 Haskell

Haskell is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. Haskell is a non-strict purely functional language based on lambda calculus, designed by representatives of the functional programming community. The motivation for Haskell was the unification of functional programming through the introduction of a standard, widespread, modern language. Haskell is a strongly typed language with a rich type system. As in all functional language, computations are performed only by expressions. Every expression has a type. Primitive data types supplied by the language include: integers, reals, characters, lists, enumerations, tuples, and various function mappings. Haskell language implementations perform static type checking prior to execution. Haskell functions are defined as mappings between parts of the type space. Composition, curried functions, lambda forms, and higher-order functions are supported. Haskell uses lazy evaluation. It also permits definition of operators as functions (operator overloading), a convenience feature that

is unusual in functional programming systems [44, 45].

2.7.1 GpH Parallel Functional Language

GpH [41] is a parallel functional language, which extends the GHC compiler of the standard non-strict functional language Haskell, with two new combinators in order to specify parallelism. GpH is a semi implicit approach. The compiler and runtime system manage most of the parallel execution. The programmer requires only to indicate those expressions that can be evaluated in parallel.

2.7.2 Parallelism in GpH.

Parallelism is introduced in GpH by the `par` combinator, which takes two arguments that are to be evaluated in parallel. A **par** expression is not restricted to its arguments; the first argument is sparked (create a thread to evaluated the first argument) while the second argument continues to be evaluated by another parallel thread. Also, GpH has a **seq** combinator which is strict on both its arguments; it evaluates its first argument to `WHNF`(Weak Head Normal Form) and then discards it and returns its second argument. An expression is in `WHNF` if and only if it has no top-level reducable expression, i.e the expression may contain inner expressions can be reduced [31]. The default evaluation degree in Haskell is `WHNF`. The **seq** combinator is needed as [14] said: first, for strict operators whose order of argument evaluation must be changed; secondly, the combinator may be used for evaluating data structures further than `WHNF`. Sometimes the **par** combinator produces too small tasks which are not useful; in this case the **seq** combinator is used to generate useful tasks for parallel execution; thirdly, it is necessary to change the behaviour of **par** to be strict in both arguments, like newpar x y = par y (seq x y).

```
par ::  a → b → b                    --parallel composition
seq ::  a → b → b                    --sequential composition

type Strategy a = a → ()             --type of evaluation strategy
using ::  a → Strategy a → a         --strategy application
using x s = s x `seq` x

rwhnf ::  Strategy a                 --reduction to weak head normal form
class NFData a where                 -- class of reducible types
      rnf ::  Strategy a             -- reduction to normal form
```

Figure 3: Basic Coordination Constructs in GpH

```
parList ::  Strategy a → Strategy [a]
parList strat [] = ()
parList strat (x:xs) = strat x `par` parList strat xs


parMap ::  Strategy b → ( a → b) → [a] → [b]
parMap strat f xs = map f xs `using` parList strat
```

Figure 4: The parList & parMap Parallel Strategies

2.7.3 Evaluation Strategies in GpH

The evaluation strategies model provided allows the programmer to split the function definition into two parts: the algorithm and the evaluation. This is achieved by using lazy higher-order functions. The lazy higher-order functions clearly separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour [41, 46, 5] .

The Strategy function specifies the dynamic behaviour required when computing a value of a given type. A strategy on a value of type `a` is a function from `a` to the nullary value `()` executed purely for effect, and the null value is returned to indicate

completion. The `using` construct applies a strategy to a Haskell expression. The basic strategy `rwhnf` reduces an expression to weak head normal form `WHNF`, the default in Haskell. The overloaded strategy `rnf` reduces an expression to normal form (NF), i.e. containing no reductions. As there are types that are not reduced to normal form in Haskell, e.g. function types, `rnf` is restricted to types that are reduced to normal form by the `NFData` class which is instantiated for all major types. Because strategies are simply functions they can be combined, or passed as parameters using standard language capabilities. Figure 3 shows the basic operation over strategies.

Data-Oriented Parallelism Strategies specifying data-oriented parallelism describe the dynamic behaviour in terms of some data structures. For example, it provides the parList function which applies the strategies to every element in parallel. Also, a `parMap` is a data parallel function which applies its function argument to every element of a list in parallel. The `strat` parameter determines the dynamic behaviour for each element of the result list. Figure 4 shows the code for both parList and parMap strategies.

2.8 GpH Compilers and Tools

There are many tools used to develop an application written in GpH. These tools and compilers are summarised in the following paragraphs as stated in [46].

2.8.1 The Hugs and GHCI Interpreter

Hugs [47] and GHCI provide an interactive environment for fast program development. They allow the programmer to experiment and debug her/his sequential program. Also, they have the ability to mix interpreted modules with compiled modules [48]. In a functional language all constructs in a program are expressions with deterministic value.

All variables have the single-assignment property, and no side-effects from calling other functions are possible. Such properties permit examining the values of certain program expressions and testing individual sub-functions in isolation. This can be done using the Hugs Interpreter. Hugs and GHCI were used to produce the initial sequential program in section 4.2.

2.8.2 The GHC Compiler and Sequential Runtime System.

GHC [49] is an optimising compiler for the non-strict purely functional language Haskell. It includes different analysis phases that supply information about the program behaviour to the optimisation phase. In GpH parallel programming, the obtained sequential optimising program is used in order to achieve parallelism. The only change required is to add strategies into sequential program. The GHC compiler was used for compiling the different sequential versions of genetic program (see Section 4.4 for more details).

2.8.3 GUM Parallel Runtime System

GUM [17] is a portable, parallel implementation of the Haskell functional language. It is message-based, and portability is facilitated by using the PVM communications harness that is available on many multi-processors. As a result, GUM is available for both shared-memory (Sun SPARCserver multiprocessors) and distributed-memory (networks of workstations) architectures. GUM uses an unmodified version of GHC to generate an optimised code. The two additional constructs `seq` and `par` specify the evaluation order and generate parallelism. GUM automatically manages many of the parallel aspects of a GpH program, including work and data distribution and distributed garbage collection. GUM's load balancing mechanism allows a high amount

of potential parallelism and distribution of the potential work in the form of sparks. Once a spark has been turned into a thread, or been activated, the thread will remain on this PE. Sparks are generated via executing the par primitive on a CPU and added to the spark pool. Initially all processors, except for the main PE, will be idle, with no local sparks available. The Idle PE sends a FISH message to a randomly chosen PE. On arrival of this message, the PE will search for a spark and, if available, send it to the requesting PE. This mechanism is usually called work stealing or passive load distribution.

2.8.4 Time and Space Profilers

The lazy evaluation mechanism in Haskell may cause some data structures not to be evaluated, or it may retain big data structures which are not used. This is called a space leak, a common problem in non-strict languages. In order to deal with this problem,the GHC [50] compiler supports a performance-tuning of the sequential program using time and space profilers. The profilers allow the programmer to assign a cost centre to any expression of the source code; thereby he/she knows the computation cost and heap usage. For example Figures 12 and 15 in Chapter 4 include the space profiles for the genetic alignment program. The time profiler allows the programmer to assign a cost centre to any expression within the functions to see its cost. For example Figure 20 in Chapter 5 includes a partial from the time profile of the final sequential version of the genetic alignment program.

2.8.5 GranSim Simulator

GranSim [6] is a highly-parameterised simulator which allows the programmer to simulate different parallel architectures. GranSim is a tool for achieving architecture-independence. By providing an idealised as well as an accurate model of parallel architectures, GranSim has proved to be an essential part of an integrated parallel software engineering environment. The idealised simulation hides all details of the underlying parallel architecture (see section 5). According to the amount of parallelism achieved from the idealised stage, the programmer takes her/his decision either to perform the realistic stage or not. Chapter 6 describes the use of the GranSim simulator to emulate specific architecture.

There are number of run-time options parameters provided by the simulator. The parameters used by the thesis are as follows:

-bP This option controls the generation of a GranSim profile. The overall activity profile shows the activity of the whole machine by separating the threads into up to five different groups, running threads, runnable threads, blocked threads, fetching threads, migrating thread

-bpn Specifies the number of processors to simulate. The value of n must be less than or equal to the word size on the machine (i.e. usually 32). If n is 0 GranSim-Light mode is enabled.

-bp: Enable GranSim-Light (same as -bp0). In this mode there is no limit on the number of processors and no communication costs are recorded.

-bln Set the latency in the system to n machine cycles. The default value is 1000 cycles.

-bmn Set the overhead for message packing to n machine cycles. This is the overhead for constructing a packet independent of its size.

2.8.6 Visualisation Tools

Visualisation tools are more important for understanding the dynamic behaviour of the parallel program. By using the visualisation tools, the log file from the simulator and GUM can be used to generate a number of graphical graphs containing information about execution of the program [48]. For example Figure 23 in Chapter 5 includes the idealised activity profile of the genetic alignment program.

Chapter 3

A Multiarchitecture Development Methodology

This chapter describes the new methodology for writing multi-architecture programs and gives a description of a program development for multiple architectures.

3.1 Overview

Parallelism in GpH is semi-explicit; only small amounts of code are required to describe the parallelism in the program. In addition, strategies allow the programmer to specify the coordination at high level, and separate the algorithm and the coordination. These properties facilitate the task of parallel programming and changing to a new architecture. Consequently, a programmer can start her/his program without any explicit parallelism, so she/he can develop and test it in a sequential environment. Then the strategies are inserted to the sequential version to produce parallel versions.

3.2 The Methodology Structure

Approximately two dozen non-trivial GpH parallel programs have been developed, for number of architectures [46, 51, 5], and Trinder and Loidl have proposed a GpH multi architecture programming methodology as result of this experience. The methodology is summarised in Figure 5, where each node is a program/virtual machine pair. The program development has two phases: an architecture-independent phase, that develops adequate parallelism on a simulated idealised machine. Experience has shown that most of the development work is done in this phase. The architecture-dependent phase tunes the parallel program for a specific architecture [46].

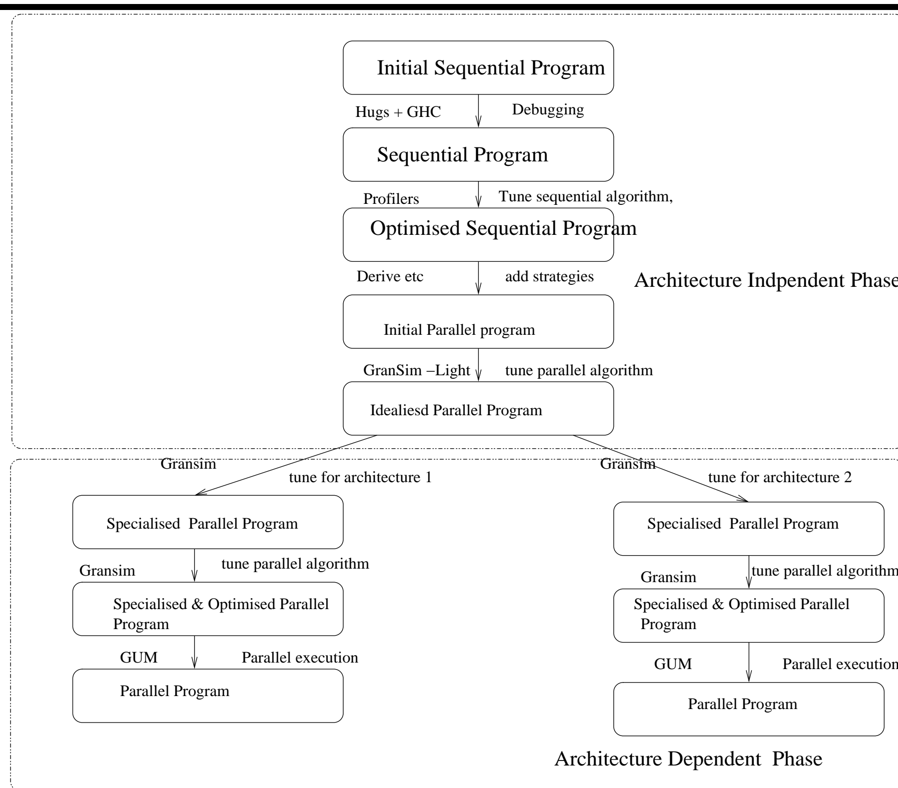


Figure 5: The Multi-Architecture Program Development Model

3.2.1 Initial Sequential Stage

In this stage the programmer writes a sequential version of the program and may need to debug it using the Hugs or GHCI interpreter [47, 50, 45]. As described in Section 2.8.1 both interpreters provide fast interactive environment development tools. The output from this stage is a correct sequential program.

3.2.2 Sequential Optimised

In this stage, profiling tools, including space and time profiles are used to obtain information about the program behaviour, including the total execution time, the allocation and residency, often itemised by individual function. Based on that information the program is tuned to produce an optimised sequential version. The output from this stage is an optimised sequential program.

3.2.3 Idealised Simulation Stage

In this stage, the evaluation strategies are added in order to expose parallelism in the program. The insertion of the strategies will be based on the information obtained from the optimised version.

The initial parallel version is measured using the GranSim [6] simulator parameterised to emulate an idealised machine with, e.g. an infinite number of processors, and zero communication costs. The parallel program will be tuned until it shows a good parallel performance.

The primary advantage of using an idealised machine is that it is known that poor parallelism is inherent, and not an artifact of some specific architecture. If good parallelism cannot be achieved on the idealised machine it cannot be obtained on any real machine. The output from this stage is an initial parallel version of the program.

3.2.4 Realistic Simulation Stage

In this stage, the parallelism is tuned for a target architecture. The tuning again uses the profiling suite, but now the simulators are parameterised to emulate the target architecture. The parameters specify details such as number of processors, message latency, thread creation overheads, all in terms of machine cycles, an abstract time measure. Typical changes during this stage are to adapt the parallelism to the characteristics of the target architecture; for example thread granularity might need to be increased to offset creation overheads and message latency. The idealised program is measured using the GranSim simulator, but here the simulator is parameterised to emulate the target machine. It is often necessary to remove some strategies from the idealised program to obtain good performance on a simulated realistic. The output of this stage is parallel program tuned for a specific architecture.

3.2.5 Target Architecture

The final stage is to measure and tune the program on the target architecture using the `GUM` runtime system and profiling tools [48]. The experiences of developing parallel programs using GpH indicate that this stage typically requires few changes [46]. Normally the simulated results are a good approximation to the parallel behaviour under `GUM` [17]. Typical changes during this stage are to adapt the I/O, or to utilise specific system calls on the target architecture. The output of this stage is a parallel program on specific architecture.

3.3 Enhancement of Architecture Independent Enhancement in GpH

The author proposes a new model involving the underlaying architecture parameters when it generates the potential parallel tasks. The parameters that may be involved are the number of available processors, system latency and the clock speed. Section 8 will describe the proposed model in detail.

3.4 Other Methodologies

There have been few robust parallel functional languages, and hence relatively few large parallel functional programs developed. As a results there are few development methodologies for parallel functional programming. Two fundamental methodologies related to functional programming that have been proposed and explored are BMF by Pepper [52] and APMs by O'Donnell [53]. Both methodologies are derivational: the parallel program is derived from a high level specification but typically, the result of the derivation is not a parallel functional program, but rather C with MPI or a parallel hardware specification. CSP introduced by Hoare provides a general skeleton for parallel programs and it allows accurate analysis of correctness and performance issues. It provides annotation which has a good interface between the communicating system and a theoretical framework [54]. CSP may be used for parallel functional programming.

The methodology described in this thesis provides a systemic manner to write a parallel program functional for different architectures. The result from methodology is parallel functional program.

Chapter 4

Sequential Implementation

This chapter describes the problem selected for implementation using the multiarchitecture development methodology, and the sequential implementation of the algorithm. It will also describe the sequential time and space tuning of the program.

4.1 Problem Description

4.1.1 A Genetic Alignment Algorithm

The program developed for several parallel architectures aligns sequences of genetic material (RNA) from related organisms and has been described in [55, 56]. The aim of creating the alignment is to study the similarities and differences in sets of sequences. An alignment of these sequences allows a biologist to extract a fairly accurate guess about how these organisms relate in the tree of evolution. The alignment of a set of RNA sequences entails lining up the sequences with corresponding sections directly above one another. In order to achieve the alignment, `indel` (for `in` inserted, or `del` deletion) are added to them [57] as shown in Figure 6. The `"-"` character in the figure represents the `indel` characters.

```
Input sequences
  AUGCGAGUCUAUGGCUUCGGCCAUGGCGGACGGCUCAUU
  AUGCGAGUCUAUGGUUUCGGCCAUGGCGGACGGCUCAUU
  AUGCGAGUCUAUGGACUUCGGCCAUGGCGGACGGCUCAGU
  AUGCGAGUCAAGGGGCUCCCUUGGGGGCACCGGCGCACGGCUCAGU
Aligned output sequences.
  AUGCGAGUCUA-----------UGG-CUU-------CGGCCAUGGCGGACGGCUCAUU--
  AUGCGAGUCUA-----------UGGACUU-------CGGCCAUGGCGGACGGCUCAUU--
  AUGCGAGUCUA-----------UGG--UUU------CGGCCAUGGCGGACGGCUCA--GU
  AUGCGAGUC-AAGGGGCUCCCUUGG-----GGGCACCGGC----GC--ACGGCUCA--GU
```

Figure 6: Input Sequences and the Aligned Output Sequences

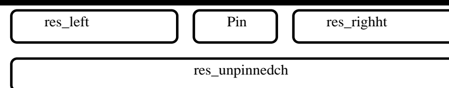


Figure 7: Final Alignment Figure.

Alignment Algorithm. The input to the program is a set of amino-acid {A,C,G,U} `Sequences`. The alignment algorithm is based on the notion of `critical subsequences`: a subsequence of a single sequence that occurs only once within the sequences. When a critical subsequence occurs in two or more sequences, the set of occurrences is called a `Pin`. To compute the `Bestpin` all the critical subsequences from each sequence must be generated, and then the critical substrings with the highest number of occurrences are selected. If more then one substring being selected as pin, the pin closest to the middle will be selected. The following steps are employed to align a set of sequences:

1. Compute a set of pins for the sequences to be aligned. Locate the best pin which has the maximum number of occurrences.

2. Connect all pinned sequences with a best pin and place it above the unpinned sequences. This results in the original sequences being divided by the best pin into three regions (left, right, and unpinned sequences). Figure 7 shows the final

alignment of the input sequences. The pinned sequences are split by the best pin and placed on top of the unpinned sequences as shown in Figure 7.

3. Recursively align the left, right and unpinned sequences.

4. Combine the pinned and unpinned alignment.

4.2 Sequential Implementation

The program consists of three main functions: `align_chunk`, `divide`, and `Bestpin`, along with auxiliary functions as depicted in Figure 8. The figure shows the dependences among the functions in the implementation.

The Align-chunk Function aligns a set of sequences (chunks) by attempting to split the chunk into three chunks using a pin: left and right pinned chunks and an unpinned chunk. These can be aligned independently and the three sub alignments are combined to produce the complete alignment. It calls `Bestpin` to extract the best pin, then calls the divide function to split the three regions as described earlier. Figure 9 shows the sequential code of the `align_chunk` function.

The Bestpin Function takes the input sequences and extracts the best pin by calling the functions placed under it: first, the `(substring_sequences)` function generates all sub strings from each sequence, because it performs an iteration over the input sequences it is called the outer loop; second, the (`Form_pin)`function computes the number of occurrences of each substring, because it performs an iteration over the substring generated from each sequence inside the `(substring_sequences)` function it is called the inner loop; third, the `Extract_max_pin` function selects the pins which have the maximum number of occurrences; fourth, `pin_average_distance` computes

the average distance of the input sequences and the distance between the selected pins and middle point of the input sequences. Figure 10 shows the sequential code of the `Bestpin` function.

The divide Function takes the input sequences and best pin and splits the input sequences using the best pin into three regions (left, right, and unpinned) by calling the `splitting_sequences` function. The `divide` function recursively calls `align_chunk` function to align the generated regions independently. The process is continued until no best pin can be found from any of three regions. The final step, the `Combine` is called to merge the alignment results from left, right and unpinned sequences. Figure 11 shows the sequential code of the divide function and calling diagram.

4.3 Alignment Example

The following example shows how the algorithm working. The following input set of sequences is given to the program.

```
[[U,C,A,G,U]
[U,C,A,G,U],
[U,C,A,U,U],
[U,C,A,U,U]]
```

The first function called is `Align_chunk` which takes the input and calls the `Bestpin`.

1. The `substring_sequences` generates all possible substrings from each input sequence as follows.

```
[[U],[UC],[UCA],[UCAU],[UCAUU],[C],[CA],[CAU],[CAUU],[AU],[AUU],[U],[UU],[U]]

[[U],[UC],[UCA],[UCAU],[UCAUU],[C],[CA],[CA,U],[CAU,U],[AU],[AUU],[U],[U,U],[U]]

[[U],[UC],[UCA],[UCAG],[UCAGU],[C],[C,A],[CAG],[CAGU],[AG],[AGU],[G],[G,U],[U]]

[[U],[U,C],[U,CA],[UC,AG],[UCAGU],[C],[CA],[CAG],[CAGU],[AG],[AGU],[G],[GU],[U]]
```

2. The `substring_sequences` calls the `From_pin` to compute the pins. The following list shows partial from Pins list with its occurrence.

   ```
   [([UC],4),([UCA],4),([UCAU],2),([UCAUU],2),([C],4)
   .
   .
   U],2),([AG],2),([AGU],2),([G],2),([GU],2)]
   ```

3. The `extract_max_pins` function extracts the pins which has maximum occurrence.

   ```
   [([UC],4),([A],4),([CA],4),([C],4),([UCA],4)]
   ```

4. The `longest_pin` function takes the output from `extract_max_pins` function and returns the longest pin, in this case `UCA.` The `pin_average_distance` function is called to compute the distance between the middle of the sequence and the pin position within the sequence. The output of `pin_average_distance` function is `[([UCA],1)].`

5. The `best_pin_pin` function takes the output from `pin_average_distance` function and returns the pin which has minimum distance from the middle: `UCA.`

6. The `Divide` function takes both the best pin and the input sequences and calls The `splitting_sequences` to split the input sequences into three sequences as follows:

   Right sequences:

   ```
   [[GU],
   [GU],
   [UU],
   [UU]]
   ```

Left sequences: `[[], [] , [], []]`, Right sequences: `[[], [] , [], []]`,

Unpinned sequences:`[[], [] , [], []]`

7. The left, right, and unpinned sequences will aligned by recursive call of the `divide` function. In this case only the right will be aligned.

   The steps from 1 to 6 will be repeated. So the second best pin is `[U]`.

   Right sequences

   ```
   [[],
   [],
   [U],
   [U]]
   ```

   Left sequences

   ```
   [[G],
   [G],
   [],
   []]
   ```

   Unpinned sequences.

   ```
   [[], [] , [], []]
   ```

   At this stage no best pin is found the `combine` function is called.

8. The `combine` function will combine the result from aligning the right sequences.

   ```
   [[GU-]
   [GU-],
   [-UU],
   [-UU]]
   ```

9. In final stage it combines all the results from left, right, and unpinned sequences.

   ```
   [[UCAGU-],
   [UCAGU-],
   [UCA-UU],
   [UCA-UU]]
   ```

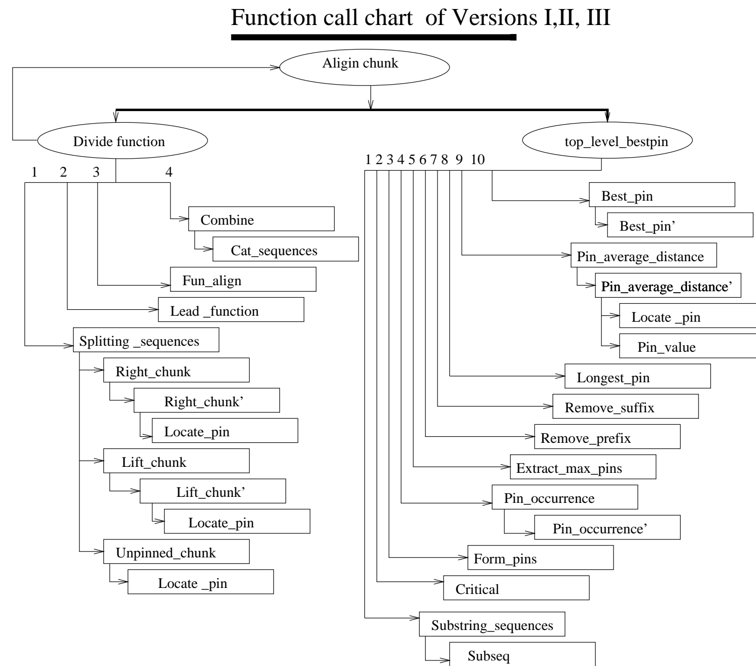


Figure 8: Functions Call Chart for Versions I,II,III.

4.4 Sequential Tuning

This section describes the steps taken to improve the sequential version of the alignment program. In fact, there is no particular rule to follow: the optimistions are presented in the order they occurred to the author. The generated heap profile which contains information about the memory usage over time is useful for detecting the causes of space leaks, when the program holds on to more memory at run-time that it needs to. Space leaks lead to longer run-times owing to heavy garbage collector activity, and may even cause the program to run out of memory altogether. From the heap profile in Figure 12 the large consumption of memory can be seen: the total allocated is 1034 Mb with maximum residency 12.250 Mb. The most expansive function as seen from the graph is the `Bestpin` function. To improve the program a series of five optimisations is made and the following sections will describe them.

```
Align_chunk ::  [Sequence] → [Sequence]
Align_chunk [] = []
Align_chunk xs = fun_align all_res
           where
                best = Bestpin xs -- Find the best pin from xs
                all_res = divide xs best -- Split and align the xs
```

Figure 9: The Align Chunk Function Sequential Code

```
Bestpin ::  [Sequence] →            -- List of input sequences.
                    Pin            --Best pin as output.
Bestpin [] = []
Bestpin xs = best_pin pins_dis
           where
                all_substring = substring_sequences xs xs
                pins = map fst( extract_max_pins all_substring)
                extract_longest_pins = longest_pin pins
                pins_dis = pin_average_distance extract_longest_pins xs
```

Figure 10: The Bestpin Function

4.4.1 Development in Versions I, II and III

In version I the `extract_max_pin` function traverses the list containing the pins with their occurrences three times in order to filter the pins holding a maximum number of occurrences. Also the `longest_pin` function traverses its given list three times in order to extract the longest pin. While in version II both functions were improved to traverse their given list just twice, Figures 13 and 14 show the modified code for both the functions.

In version III the foldr high order function was employed to improve the `extract_max_pin` and `longest_pin` functions instead of using the accumulative variable.

```
divide ::  [Sequence] →       -- List of input sequences
           Pin →          -- Best pin
           [Sequence]       -- List of aligned sequences.
divide [] [] = []
divide xs [] = xs   -- this is represent the basic alignment to the sequence
divide xs pin = (combine pin res_left res_right res_unpinch )
           where
               (rightch,leftch,unpinch1) = splitting_sequences pin xs
               unpinch = lead_function pin unpinch1
               res_unpinch = align_chunk unpinch
               res_right = align_chunk rightch
               res_left = align_chunk leftch


combine ::  Pin → [Sequence] → [Sequence] → [Sequence] → [Sequence]
combine pin left_seqs right_seqs unpinned_seqs
                = ( zipWith ( cat_sequence pin) left_seqs right_seqs)
                     ++ unpinned_seqs
           where
                cat_sequence ::  Sequence→ Sequence→ Sequence → Sequence
                cat_sequence pin ls rs = ls ++ pin ++ rs
```

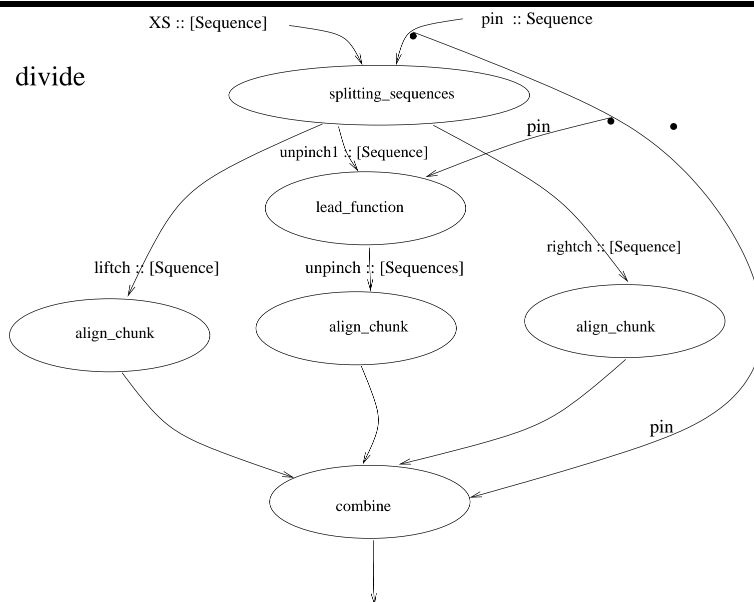


Figure 11: Divide and Conquer Sequential Code and Diagram.

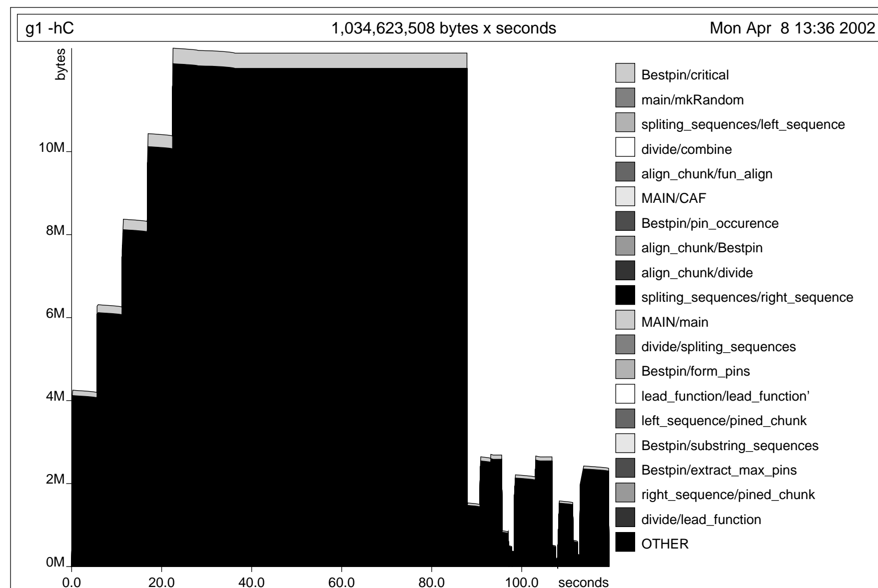


Figure 12: Heap Profile of Initial Version.

```
-- This old code for extract_max_pins function
extract_max_pins ::  [(Pin,Int)] → [Pin]
extract_max_pins [] =[]
extract_max_pins xs = map fst ( filter (λ (p,n)→ n== max_num) xs )
           where
                max_num= maximum (map snd xs)

-- A new code for the extract_max_pins function
extract_max_pins ::  [(Pin,Int)] → [(Pin ,Int )]
extract_max_pins [] =[]
extract_max_pins ((p,n):xss) =foldr (extract_max_pins') [(p,n)] xss

extract_max_pins' ::  (Pin ,Int)→ [ (Pin,Int)] → [(Pin,Int)]
extract_max_pins' (p,n) aa_pin@((p',n'):_)
                | n' > n = aa_pin
                | n' == n = aa_pin ++ [(p,n)]
                | otherwise = [(p,n)]
```

Figure 13: Old and New Code of Extract_max_pin Function.

```
-- The old code for longest_pin function
longest_pin ::  [Pin]→[Pin]
longest_pin [] = []
longest_pin xs = longest_pin' m xs
           where
                m = maximum (map length xs )
longest_pin' m [] = []
longest_pin' m ( x:xs)
                | m == length x = x :longest_pin' m xs
                | otherwise = longest_pin' m xs

-- The modified code for longest_pin function
longest_pin ::  [Pin]→ [Pin]
longest_pin [] = []
longest_pin (xs:xss) =foldr (longest_pin') [xs] xss
longest_pin' ::  Pin → [Pin] →[Pin]
longest_pin' pin xs@(pin':_)
                | length pin' > length pin = xs
                | length pin' == length pin = (pin:xs)
                | otherwise = [pin]
```

Figure 14: Old and New Code of Longest_pin Function.

The modifications made in versions II and III do not give a big improvement in consumption of memory. The heap profile obtained from both versions is similar to figure 12, therefore is not included here. The next step to improve the program is to eliminate some intermediate data structure.

4.4.2 Development of Versions IV and V

The changes made to produce IV and V are intended to eliminate intermediate data structures; e.g. an important optimisation is to eliminate the unpinned substrings at an earlier stage. In other words, when the substrings are generated from a single sequence the program computes the pin substrings before it generates the substring from other sequences. This means only the pin substrings are carried to the next stage. As a

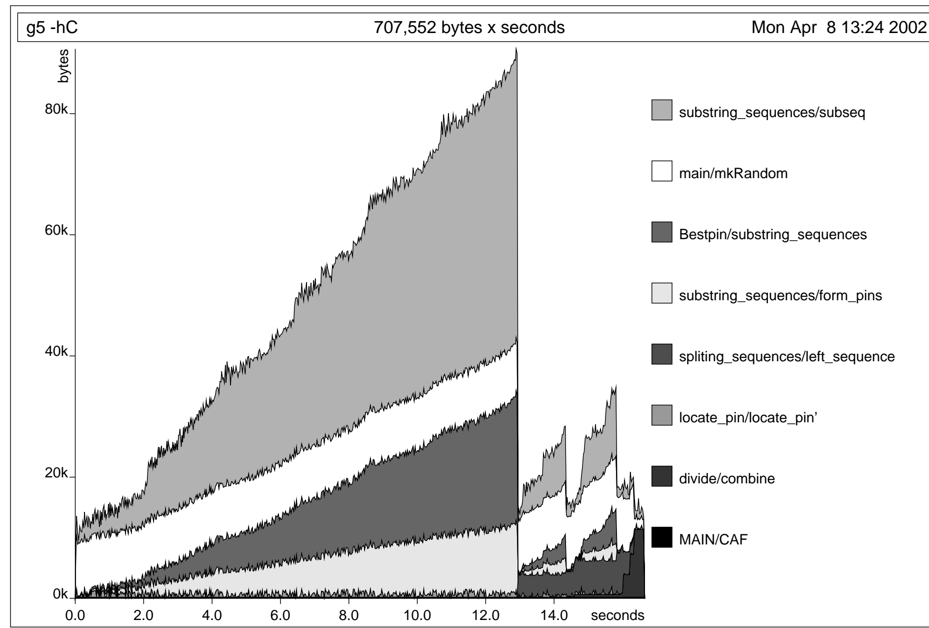


Figure 15: Heap Profile of the Final Sequential Version.

result of the above modification, the total memory allocated dropped to 707.552 Kb with maximum residency 122.70 Kb. Also the total execution time was reduced to 18.92 seconds, comprising 17 seconds real execution time (MUT) of reduction and 2 seconds of garbage collection (GC). Figure 15 shows the heap profile for the final tuning of the sequential stage. Figure 16 shows all modifications are made on the code of `From_pin` in versions IV & V.

4.4.3 Version VI using a Finite Map

In the final version a finite map is employed to search and eliminate the duplicated substring generated from the single string, and to eliminate the unpinned substring from the critical substring generated by the above step. The code is shown in Figure 17. This version gave some improvement in code execution time compared with versions

```
-- The old code for form_pins function
form_pins ::  [[SubSequence]] -> -- All substrings for input sequences
                [Pin] -- List of all Pins
form_pins [] = []
form_pins (xs:xss) = [x | x <- xs , or [ x `elem` ys | ys <- xss] ]

-- The New code for form_pins function
form_pins ::  [SubSequence] → [Sequence ]→ [(Pin,Int )]
form_pins [][] = []
form_pins[] ys = []
form_pins(x:xs) ys
                | num > 1 = (x,num):  form_pins xs ys
                | otherwise = form_pins xs ys
           where
                num = form_pins' x ys
form_pins' ::  SubSequence → [Sequence] → Int
form_pins' [] [] = 0
form_pins' m [] = 0
form_pins' m (x:xs)
      |((check_for_snd_appears == Nothing ) &&
      (check_for_appears /= Nothing )) =
      1+ form_pins' m xs
      |otherwise = form_pins' m xs
  where
      check_for_appears = locate_pin m x
     where_pin_appears = pin_value(check_for_appears)
     reset_of_sequence = drop (where_pin_appears + length m ) x
                check_for_snd_appears = locate_pin m reset_of_sequence
```

Figure 16: The Modified Form_pin function of Version IV & V.

IV and V. However the garbage collection time increased from 1.93s to 72.96s and the residency rose from) 0.123 Mb to 13.4 Mb. The rise is the result of the strict map construction generation that takes the generation of all substrings from all input sequences, and the holding of the unpinned critical substring until the stage where the pins are computed.

	time				*memory*	
Version	Mut	GC	total time	% of GC time	max residency	total allocate
I	120.1s	104.0 s	224.13 s	46.4	12520 Kb	141.8 Mb
II	121.28 s	100.8 s	222.10 s	45.4	12520 Kb	141.8 Mb
III	119.08 s	100.8 s	219.92 s	45.9	12520 Kb	141.7 Mb
IV	37.27 s	6.52 s	43.80 s	14.9	123 Kb	1,349.0 Mb
V	16.98 s	1.93 s	18.92 s	10.2	123 Kb	369.5 Mb
VI	12.96 s	72.96 s	85.93 s	84.9	13400 Kb	128.05 Mb

Table 1: Sequential Profiling Summary

4.4.4 Sequential Optimisation Discussion

Table 1 summarises most of the measurements of the sequential program versions, and the following observations are made: the time required to execute the code (Mut) reported in the second column and the garbage collect (GC) time reported in the third column.

1. The first three versions perform massive memory allocation and have high residency, resulting in long execution times, e.g 141.8 Mb and 120.1s, respectively.

2. Good performance is obtained from version IV of the program compared with version III. From Table 1 the memory residency dropped to 123 Kb, and runtime improved to 43.80, a factor of 5.02. This is the result of eliminating the unpinned strings at an earlier stage.

3. Version V further improved the execution time to 18.92s, with the same residency (0.123Mb), but less allocation 369.5Mb. The explanation for the improvement is that the old version of the `locate_pin` function takes a single sequence and pin and finds the position of the pin in the sequence by generating a substring which is equal to the length of the given pin. Then it compares this substring with the pin; if they are equal it returns to the position, otherwise the function drops one character from the sequence and repeats the process again. In the new version each time the program drops one character from the sequence and checks if the

pin is prefixed from it or not. If so it just returns to its position, otherwise it scans the rest of the sequence.

4. Version VI attempts to improve the execution time by introducing a finite map, but the residency rose from 123 Kb to 13.4 Mb. The rise is the result of the strict map construction generation that takes the generation of all substrings from all input sequences, and the holding of the unpinned critical substring until the stage where the pins are computed.

From the figures in Table 1 and the above discussion it may concluded that the most suitable version to parallelise is version V, and the next chapter describes this.

```
-- Old code for the modified functions
-- Sequential code for substring_sequences function
substring_sequences ::  [ Sequence] → [ Sequence] → [(Sequence,Int )]
substring_sequences [] [] = []
substring_sequences [] ys = []
substring_sequences (x:xs) ys = nub res1
           where
                res = subseq x
                res1= form_pins res ys ++ substring_sequences xs ys
form_pins ::  [SubSequence] → [Sequence ] → [(Pin,Int )]
form_pins [][] = []
form_pins[] ys = []
form_pins(x:xs) ys
               | num > 1 = (x,num):  form_pins xs ys
                | otherwise = form_pins xs ys
           where
                num = form_pins' x ys

-- New code for the modified functions
substring_sequences [] = []
substring_sequences (x:xs) =
           critical_substring++ substring_sequences xs
      where
      all_substrings = subseq x
      fm_of_substring = list_of_substring_fm all_substrings
      critical_substring = critical_function fm_of_substring
-- This is to filter substrings which occurrence once in FM.
critical_function fm =
           filter ( x -> case lookupFM fm x of Just n -> n==1)
           (keysFM fm)
list_of_substring_fm ::  [SubSequence ] -> FiniteMap SubSequence Int
list_of_substring_fm [] = emptyFM
list_of_substring_fm xs = addListToFM_C (+) emptyFM [(x,1) | x<-xs]
form_pins ::  FiniteMap SubSequence Int -> [(Pin ,Int)]
form_pins ys = pins_in_list
           where
           list_of_cri_substrings = critical_function ys
           pins_in_fm = delListFromFM ys list_of_cri_substrings
           pins_in_list = fmToList pins_in_fm
```

Figure 17: The Modified Functions to implement the Finite Map (Version VI).

Chapter 5

Idealised Measurement

5.1 Introduction

There are several sources of parallelism in the genetic alignment program and this chapter will describe the five parallel versions of the program developed using them.

The performance of each version from the program is measured on the GranSim simulator parameterised to emulate an idealised machine with zero communication costs and an infinite number of processors. The input data in each case is a set of 6 sequences containing 20 amino acids. For the last three versions a chunk of size 30 is used.

```
divide xs ys = (combine pin res_left res_right res_unpinch )
               `demanding` strategy
          where
                                 .
                .
                .
               strategy =rnf res_left `par`
                     rnf res_right `par`
                     rnf res_unpinch
```

Figure 18: The Strategies Required for Parallel Divide Function

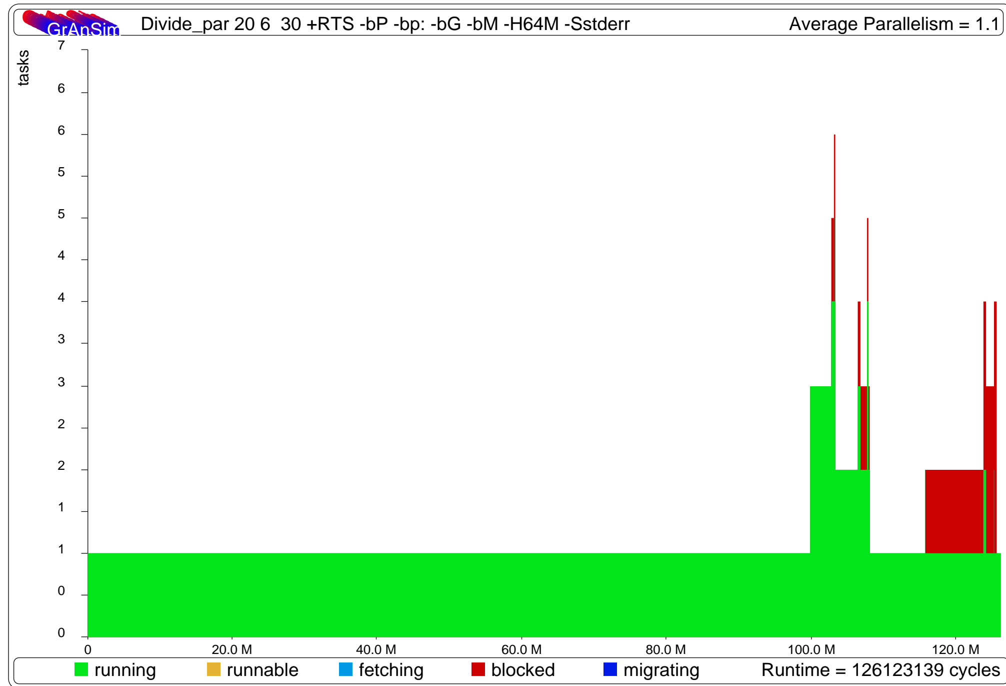


Figure 19: Idealised Simulated Profile of Version I

5.2 Version I: Divide-and-Conquer

Parallelism is initially introduced using a divide-and-conquer paradigm: the alignment of the left, right and unpinned chunks is independent so they can be evaluated in parallel, as shown in Figure 11. The required strategy for parallelising the `divide` function is shown in Figure 18.

The corresponding activity profile is presented in Figure 19 and shows the execution time on the X-axis and the number of tasks on the Y-axis. The tasks are separated into five classes, depending on their state: running if they are executing (green), runnable if they could be executed if a processor becomes ideal (yellow), blocked if they wait for data under evaluation (red), fetching if they are retrieving data from another processor (light-blue), and migrating if they are retrieved from another processor (dark-blue) [58] [46]. Figure 19 shows a small number of parallel tasks. This is owing to the long

```
Mon Apr  8 13:24 2002 Time and Allocation Profiling Report  (Final)
           g5 +RTS -pT -hC -sstderr -RTS 6 90

COST CENTRE          MODULE     %time %alloc
locate_pin'          Main        76.9   78.1
con2tag_Aminoacid#   Main        14.5    0.0
GC                   GC          11.3    0.0
form_pins'           Main         3.7    1.0
subseq               Main         2.9   20.6
substring_sequences  Main         1.2    0.0
                                                    individual     inherited
COST CENTRE              MODULE     entries  %time %alloc   %time %alloc

MAIN                     MAIN             0     0.0    0.0     100.0 100.0
 main                    Main             1     0.0    0.1     100.0  99.8
align_chunk            Main              55     0.0    0.0     100.0  99.8
   Bestpin               Main            44     0.0    0.0     100.0  99.7
     substring_sequences  Main          141      1.2    0.0    100.0  99.7
      form_pins           Main        61303     0.0    0.0      95.8  79.1
       form_pins'         Main       326287     3.7    1.0      95.8  79.1
          locate_pin        Main     530162    0.5    0.0      91.7  78.1
            locate_pin'      Main      23476420 76.9   78.1     91.2  78.1
     subseq              Main          2132       2.9   20.6      2.9  20.6
```

Figure 20: A Partial from Time Profile of the Final Sequential Version.

initial sequential segment caused by `Bestpin`, which occupies about 77 percent of the runtime. From figure 20 it can be seen that the `Bestpin` function is called before the divide function. Moreover the sequential time profiling in Figure 21 shows that the `locate_pin` function called from `Bestpin` function consumes the most execution time(77%).

By Amdahl's Law [59] the sequential component of this version of the program limits the speedup that can be achieved even under ideal conditions to $\frac{100\%}{77\%} = 1.29$.

5.3 Version IIa: Parallelising Substring Sequences

This version parallelises the outer loop of the `Bestpin` function described in Section 4.2 using a data parallel style. More specifically the `Par_substring_sequences` function

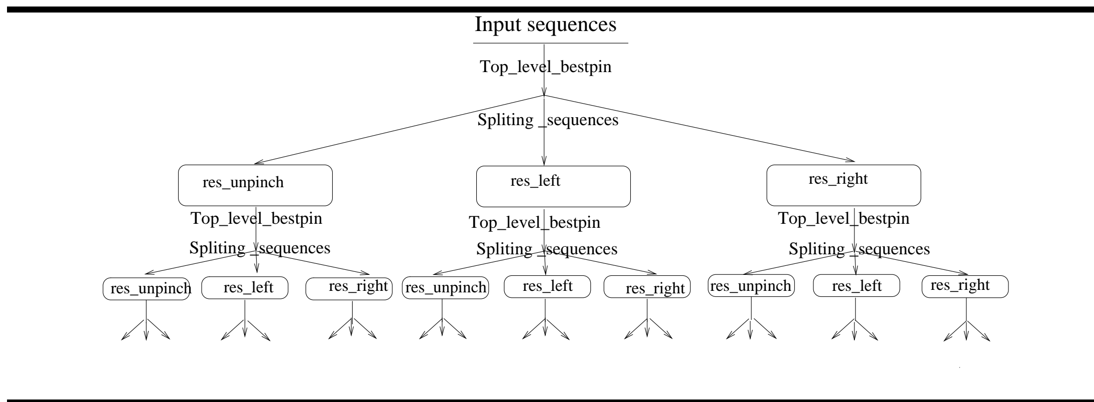


Figure 21: Divide and Conquer Process Diagram for Divide Function

uses `parMap` to the map `substring_sequences` function over the input sequences in parallel. The parallelisation is not just replacing `map` with `parMap`; it needs to modify the `substring_sequences` function so the `parMap` can used. Figure 22 compares the sequential code with the modified parallel code for `substring_sequences`, and also shows the inserted function. Figure 23 shows that six running tasks are generated by the `parMap` function. In fact the number of generated tasks depends on the number of input sequences. The speedup obtained from this version is 4.2 which indicates that parallelising functions under `substring_sequences` is a good approach to improve the execution time.

5.4 Version IIb: Parallelising Form_pin

This Version parallelises the inner loop described in Section 4.2. A new `par_from_pin` function is inserted and the `From_pin` function was modified to be executed in parallel over the supplied list. Figure 24 shows the sequential version and the parallel version of the `From_pin` function. Table 2 shows that the total amount of work is increased owing to the empty returned pair if the substring is not a pin. However, the speedup was increased to 6.9. Figure 25 shows the sequential segment at the end of each recursive

```
-- Sequential code for substring_sequences function
substring_sequences ::  [ Sequence] → [ Sequence] → [(Sequence,Int )]
substring_sequences [] [] = []
substring_sequences [] ys = []
substring_sequences (x:xs) ys = nub res1
          where
               res = subseq x
               res1= form_pins res ys ++ substring_sequences xs ys

-- Parallel code of substring_sequences function
par_substring_sequences ::  [Sequence ] → [Sequence ] → [(Pin,Int )]
par_substring_sequences xs ys =
                    foldr (++) []
                    (parMap rnf (substring_sequences ys) xs)
substring_sequences ::  [ Sequence]→ Sequence → [(Sequence,Int )]
substring_sequences ys x = nub res1
          where
               res = subseq x
               res1= form_pins res ys
```

Figure 22: Sequential and Parallel Code of Substring Function.

call of `divide`; an attempt will be made to avoid this in the next alternative.

5.5 Version IIc: Parallelise Both Outer and Inner Loops

This version combines inner and outer loop parallelism, i.e from both version IIa, version IIb. There is not much difference in the profile between version IIb and IIc, so it is not included.

5.6 Version III: Clustering on Parallel Form_pin

This version includes all previous sources of parallelisation, (i.e versions I, IIa, IIb, IIc); also, a clustering function was applied to the input list supplied to the `form_pin` function. The clustering function breaks the given list into convenient sized chunk and

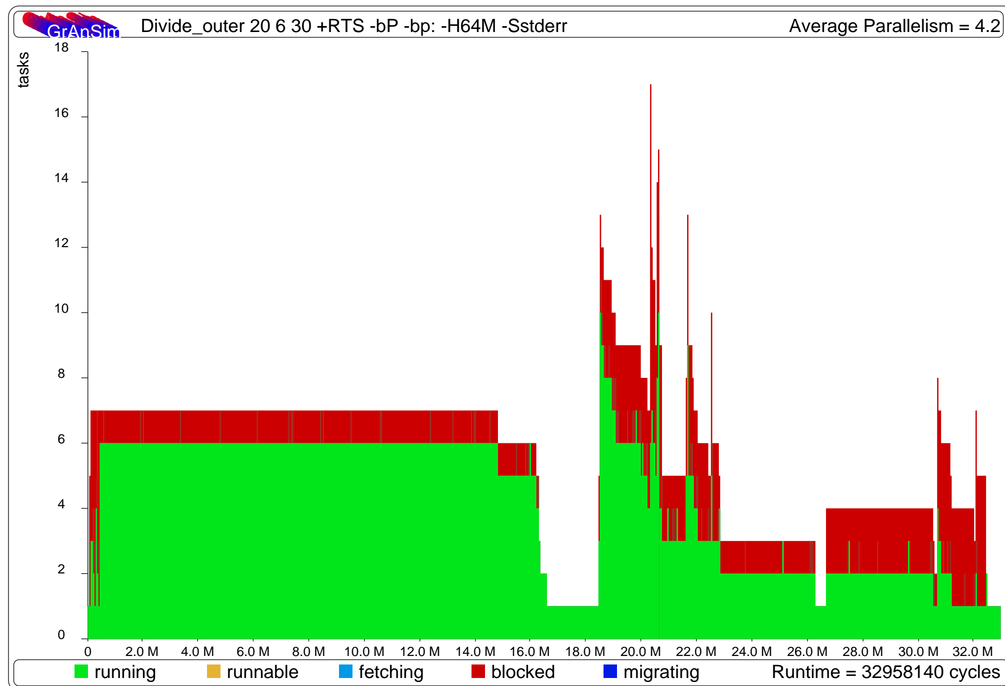


Figure 23: The Idealised Activity Profile of Substring Sequences Function (IIa).

Figure 26 shows the code. The implementation of the `parMap` function on a collection of data such as a big list often yields very fine task granularity. Clustering is one way to improve the task granularity and data locality by introducing fewer tasks, each operating on a closely-related subset of the collection [51]. As shown in Figure 27 the utilisation on the system was improved, but there is still a sequential part which needs to be eliminated.

5.7 Version IV: Parallelise all maps

This version modifies the previous version by replacing all map functions with `parMap`. In other words in this version all intermediate functions called by `Bestpin` and `divide` function are parallelised. Figure 28 shows the activity profile for the `parMap` version of the program. From the graph the improvement in the system utilisation can be seen;

```
-- Sequential code From_pin Function
form_pins ::  [SubSequence] → [Sequence ] → [(Pin,Int )]
form_pins [][] = []
form_pins[] ys = []
form_pins(x:xs) ys
              | num > 1 = (x,num):  form_pins xs ys
               | otherwise = form_pins xs ys
          where
               num = form_pins' x ys

-- Parallel code From_pin Function
par_form_pin ::  [Sequence ] → [SubSequence] → [(Pin,Int )]
par_form_pin xs ys = nub( parMap rnf ( form_pins xs ) ys)
form_pins ::  [Sequence ] → SubSequence → (Pin,Int )
form_pins ys x
               | num > 1 = (x,num)
               | otherwise = ([], 0)
          where
               num = form_pins' x ys
```

Figure 24: Sequential and Parallel Code of From_pin Function (Inner Loop).

up to 45 processors are used.

5.8 Version V: Parallel all foldr

The final version modifies version IV by adding the parallelised fold function used by `extract_max_pin` function. A new strategic function called `parfoldList` was defined to execute foldr function in parallel. Figure 29 shows the code for the above modification.

5.9 Idealised Optimisation Discussion

The results obtained from the idealised parallel versions are summarised in Table 2.

The maximum idealised speedup was obtained from versions IV and V, 21.5 and 21.9 respectively. Versions IIb, IIc, and III give more modest speedups. The table also

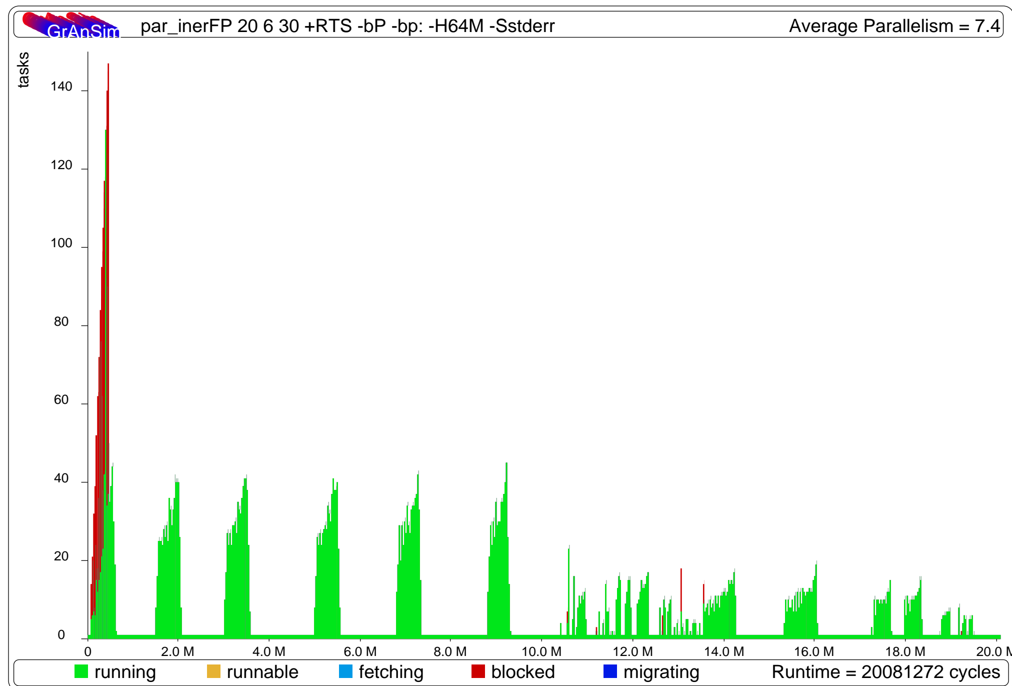


Figure 25: The Idealised Activity Profile of From pin Function (IIb)

shows the increasing number of small tasks from 19 tasks in version I to 785 in version V. There is also a small increase in total work for both versions IIb and IIc. This is owing to the fact that the `Form_pin` function operates on each element in the list in parallel and returns an empty tuple if the substring is not a pin, while in the other versions if the substring is not a pin the function does not return anything.

The most important observation from the idealised measurements is that a programmer can parallelise every point in the program even it generates small tasks, and still some speedup can be achieved. This is clearly seen from Table 2 and activity profiles from the different versions.

Figure 30 shows the overall activity profile of version V of the program. From the graph it can be seen that, for this input data, the idealised machine could utilise approximately 45 PEs. This version is the best idealised parallel version.

```
cluster ::  Int → [SubSequence] → [[SubSequence]]
cluster n [] = []
cluster n xs = take n xs :  cluster n (drop n xs)

substring_sequences ::  Int → [ Sequence] → Sequence → [(Sequence,Int)]
substring_sequences n ys [] = []
substring_sequences n ys x = nub res1
           where                 subseqlist = subseq x
                cluslist = cluster n subseqlist
                res1= par_form_pin ys cluslist
```

Figure 26: Cluster Function and Modified Substring Function.

Program	Average Parallelism	Speedup	Total Runtime (Mega cycles)	Total Work (Mega cycles)	Generated Tasks	Avg. Task Leng. (Mega cycles)
Seq	1.0	1.0	139.6	139.6		
I	1.1	1.1	126.1	138.7	19	7.3
IIa	4.2	4.2	32.9	138.1	94	1.4
IIb	7.4	6.9	20.0	148.0	3542	0.041
IIc	13.6	9.5	14.7	199.9	3583	0.052
III	16.7	16.8	8.3	138.6	275.0	0.503
IV	21.1	21.5	6.5	137.1	381.0	0.35
V	**21.9**	**21.8**	**6.4**	140.1	785.0	0.177

Table 2: Idealised Simulation Input : 20 6 30.

The next chapter describes the GranSim simulation of the different parallel versions on two architectures. The GranSim will parameterised to emulate both of these.

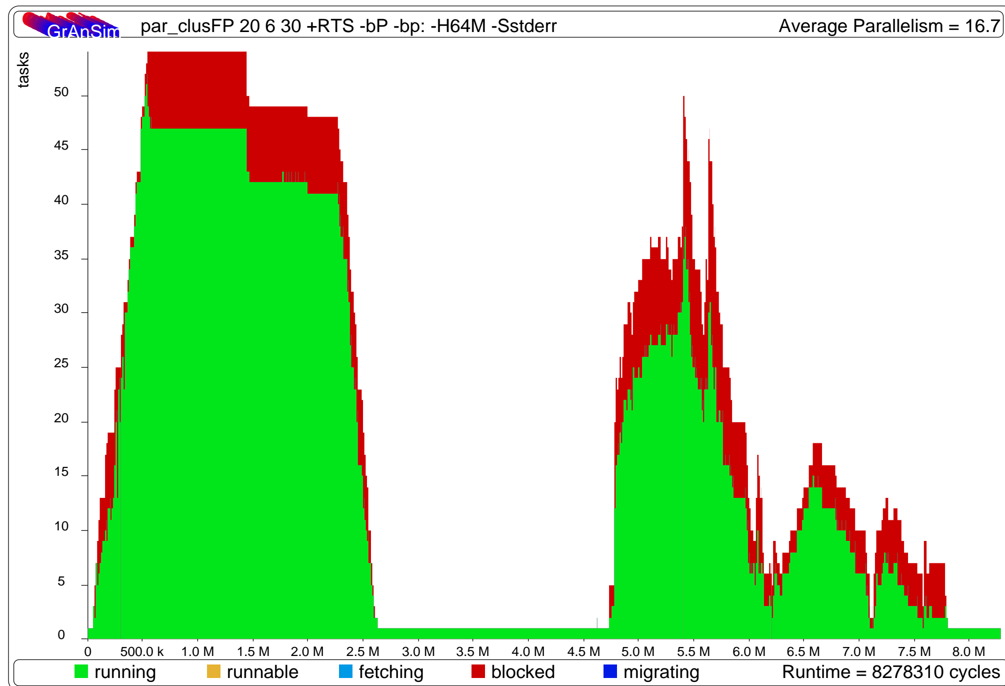


Figure 27: The Idealised Activity Profile for Clustering version (III).

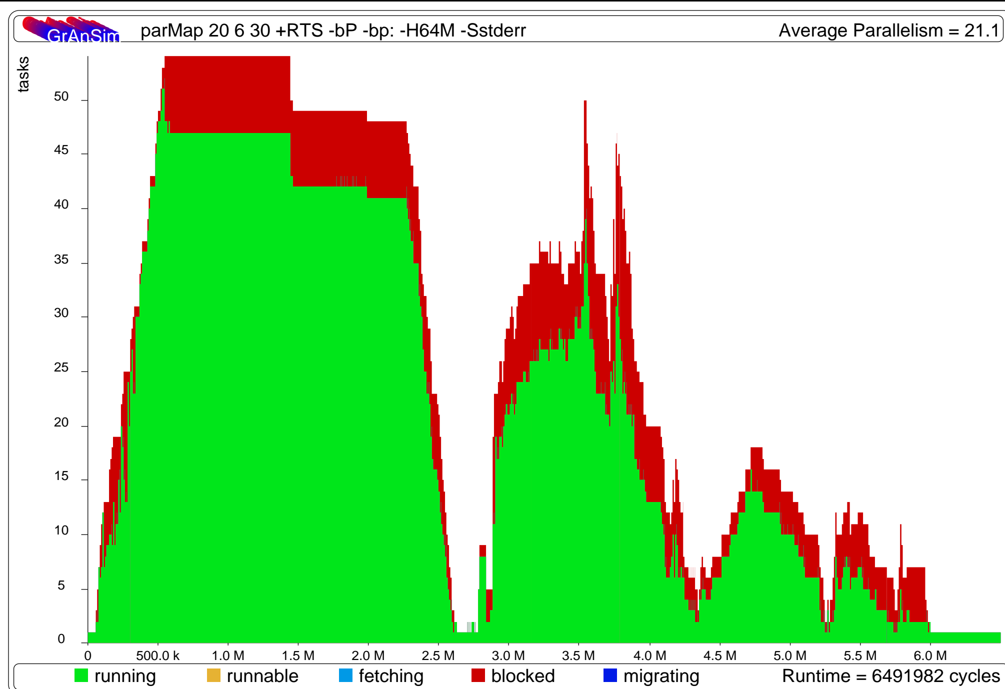


Figure 28: The Idealised Activity Profile for parMap version IV

```
extract_max_pins ::  [(Pin,Int )] → [ (Pin ,Int )]
extract_max_pins [] =[]
extract_max_pins ((p,n):xss) =
                parfoldList (extract_max_pins’) [(p,n)] xss

parfoldList ::  NFData a → (a → [a]→[a]) → [a]→ [a] → [a]
parfoldList f z [] = z
parfoldList f z (x:xs) = f x ys ‘sparking‘ rnf ys
           where
                ys = parfoldList f z xs
```

Figure 29: New ParfoldList and Extractmaxpins Function.

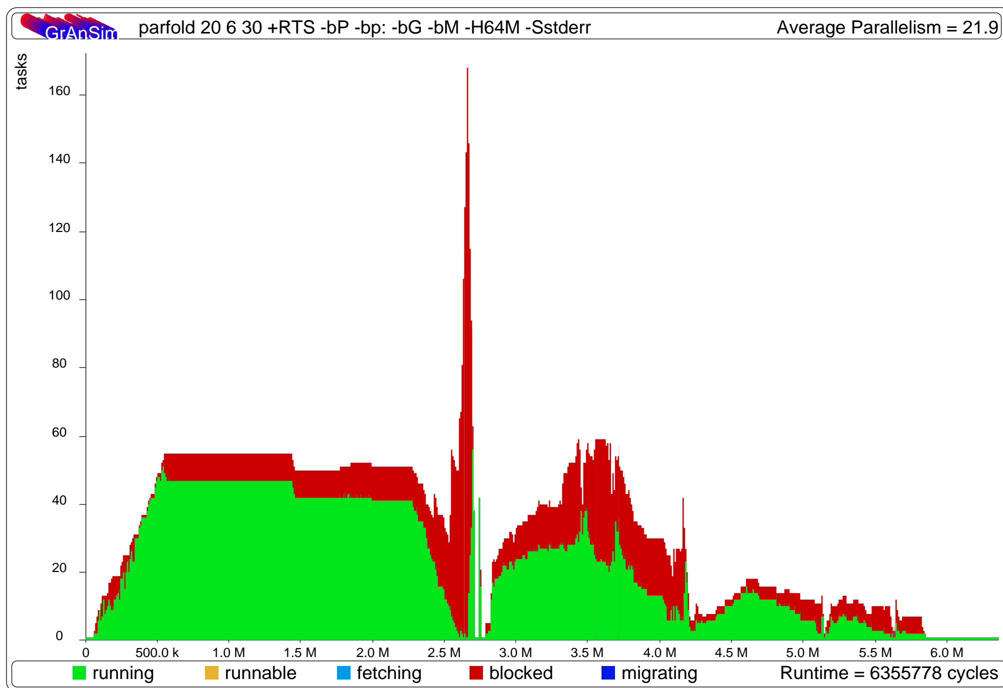


Figure 30: The Idealised Activity Profile for Version V Input 20 6 30

Chapter 6

Two Simulated Architectures

The methodology in Chapter 3 indicates that architecture dependent tuning starts with simulating the target architecture. In this experiment, the program is tuned for two different architectures: a 32 processor Beowulf cluster and a 4-processor Sun SMP. The architectures are simulated by parameterising (-bp, -bln ,and -bmn see section 2.8.5) GranSim with key architectural properties, most important of which are the number of processors, the time to pack a message for transmission, and the communication latency. The last two properties are measured in clock cycles of the given processor.

6.1 Beowulf Simulation

The target machine is a 32-node 530MHz Pentium III Beowulf cluster connected by fast Ethernet switch. To determine the GranSim parameters accurately the required parameters were measured using simple programs. For example the PE to PE communication latency was measured as 142 μs under PVM 3.4.2, so for the 530 MHz processor the GranSim latency is 142*530 = 753 kcyc. Likewise the packing time is measured as 21 μs which gives 11 kcyc. Table 3 summarises the results of the different

Program	Average Parallelism	Speedup	Total Runtime (Mega cycles)	Total work Work (Mega cycles)	Generated Tasks	Avg. Tasks Leng. (Mega cycles)
Seq	1	1	139.6	139.6	–	–
I	1.1	1.1	127.2	139.9	19	7.4
IIa	**1.9**	**1.7**	80.8	153.5	94	1.47
IIb	1.2	0.2	708.9	850.7	3183	0.04
IIc	2.2	0.9	143.7	316.1	3201	0.041
III	2.0	0.9	150.8	301.6	275	0.506
IV	1.9	0.8	168.2	319.6	381	0.362
V	2.1	0.8	162.0	340.2	785	0.178

Table 3: Realistic 32-PEs Beowulf Simulation Input: 20 6 30

versions of the Genetic alignment program with problem size 20 6 30. It should be remembered that 6 represents the number of sequences, 20 represents the length of each sequence, and 30 represents the chunk size.

6.2 Sun SMP Simulation

The target machine is 4-processor Sun SMP with a clock speed of 250 MHz connected by shared memory bus. The latency under PVM layer between nodes has been measured as 109 μs which is equivalent to 27.5 Kcyc, and the packing cost as 22 μs which is equivalent to 5 Kcyc. The results of the realistic Sun SMP simulation of the Genetic program are summarised in Table 4.

Program	Average Parallelism	Speedup	Total Runtime (Mega cycles)	Total work Work (Mega cycles)	Generated Tasks	Avg. Tasks Leng. (Mega cycles)
seq	1	1	139.6	139.6	–	–
I	1.1	1.1	126.6	139.2	19	7.4
IIa	**2.1**	**1.9**	70.6	148.2	94	1.47
IIb	1.3	0.3	413.3	537.2	3542	0.042
IIc	2.9	1.4	97.6	282.0	3201	0.046
III	3.5	1.8	77.2	270.2	275	0.51
IV	3.3	1.7	81.4	268.6	381	0.362
V	3.4	1.7	81.6	277.4	785	0.178

Table 4: Realistic 32-PEs Sun SMP Simulation Input: 20 6 30

6.3 Discussion of Simulation Results

6.3.1 Idealised Simulation vs Realistic Simulation

Comparing the idealised and realistic simulations, Tables 2 3 and 4, the following observations were made:-

- For these small input sizes the speedup attained and utilisation of each architecture is extremely poor.
- The number of generated tasks is similar in all three simulations because most parallelism is in flat (data parallelism) rather than hierarchical (divide & conquer).
- The simulated Sun SMP does more work than the idealised simulation, and the simulated Beowulf does more work than the simulated Sun SMP. This reflects the increasing latencies of the architectures.
- Both simulated machines give much worse speedups than the idealised machines, with the simulated Beowulf being slightly worse than the simulated Sun SMP. This is caused by the latency of each architecture in realistic simulated machines.
- Increasing the number of generated tasks always gives a better speedup in an ideal machine, but this not the case on realistic machines, because of the communication and tasks management overheads introduced in the realistic simulation.
- Figures 31 and 32 show the differences between the activity profiles for the program versions on the idealised machine and the simulated Beowulf. There is a similarity between the idealised and the simulated activity profiles for versions I; this is because version I generate a small number of parallel tasks. In contrast with other versions, there are differences in activity profiles; the most significant

differences come from the communication cost of the simulated machine. Moreover, the larger runnable threads seen from the graphs are the result of the limited of number of PEs in the realistic simulation.

- From Figures 31 and 32 it is clear that, as expected,the idealised simulation does not predict realistic simulation. This because the realistic includes realistic overhead costs, especially communications. However, the idealised stimulation does allow the separation of algorithm and architecture concerns: a program that fails to deliver good parallel performance on a simulated idealised machine, cannot deliver good performance on any real architecture.

6.3.2 Beowulf Simulation vs Sun SMP Simulation Comparison

The following observations were made in comparing the Beowulf and Sun SMP simulations in Tables 3 and 4:-

1. Versions I and IIa of the program have similar behaviour on both architectures. This is because they generate a small number of large tasks compared with other versions.

2. Separate experiments show that better speedups could obtained for both simulated architectures with large input sizes, but the execution time and disk space on the simulation platform limit the input size for systematic experiments. Figure 34 shows the speedups obtained from executing the different versions of the program on both architectures with varying of PEs. Even with a small input size the maximum speedup is 2.5 on the simulated sun SMP and 1.7 on the simulated Beowulf, both for version IIa.

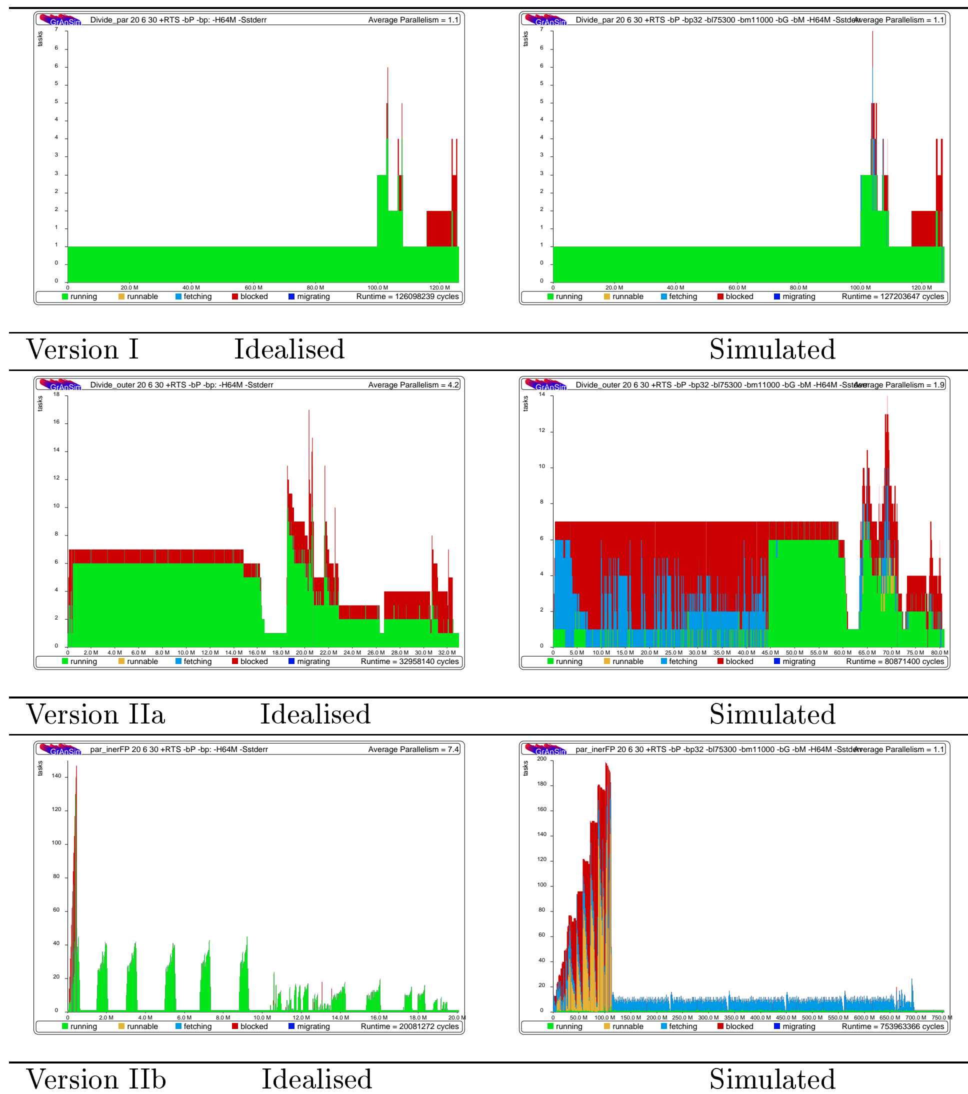


Figure 31: The Activity Profile for Idealised vs Simulated Beowulf(Version I, IIa, and IIb)

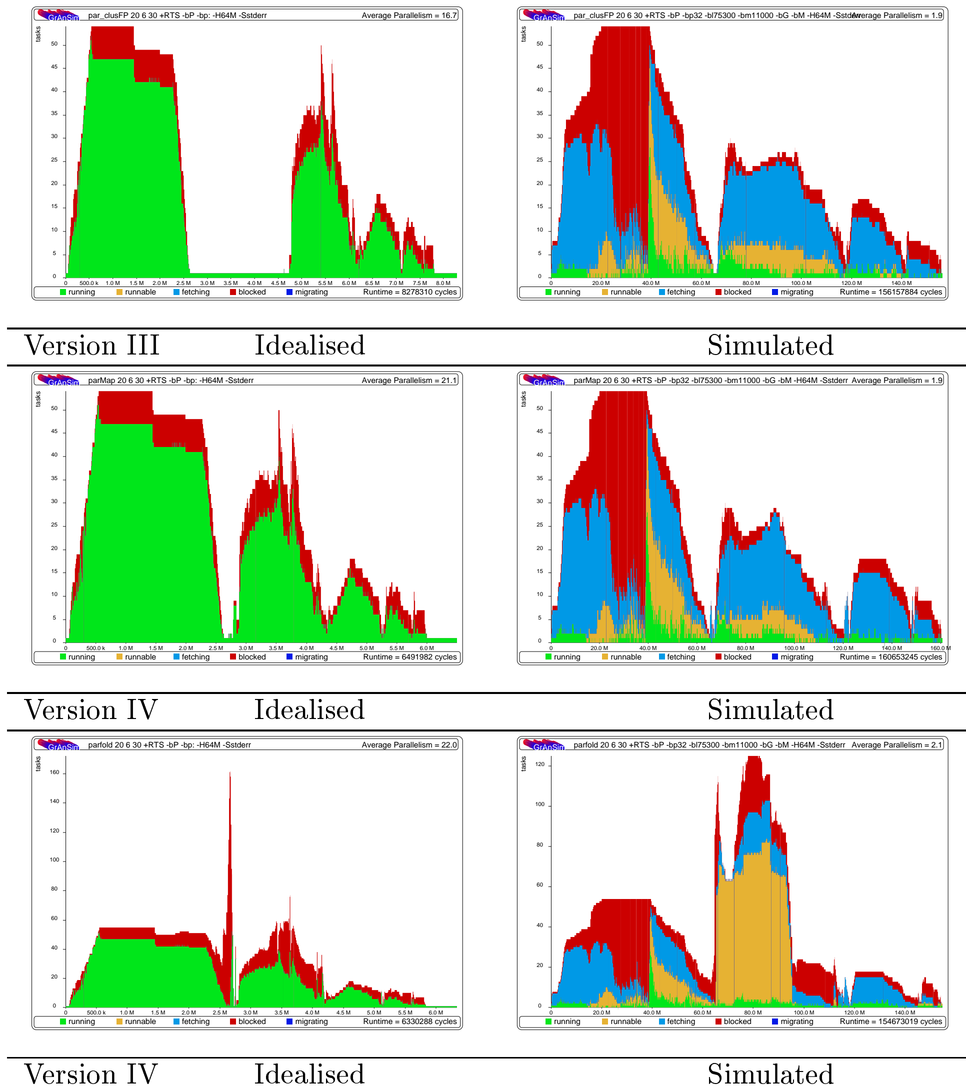
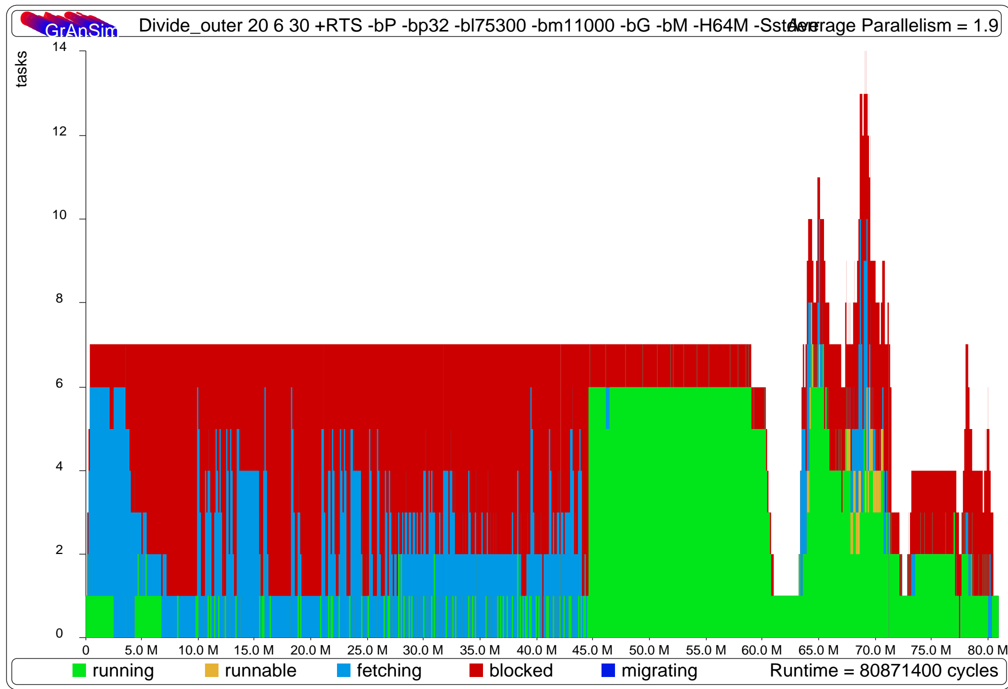
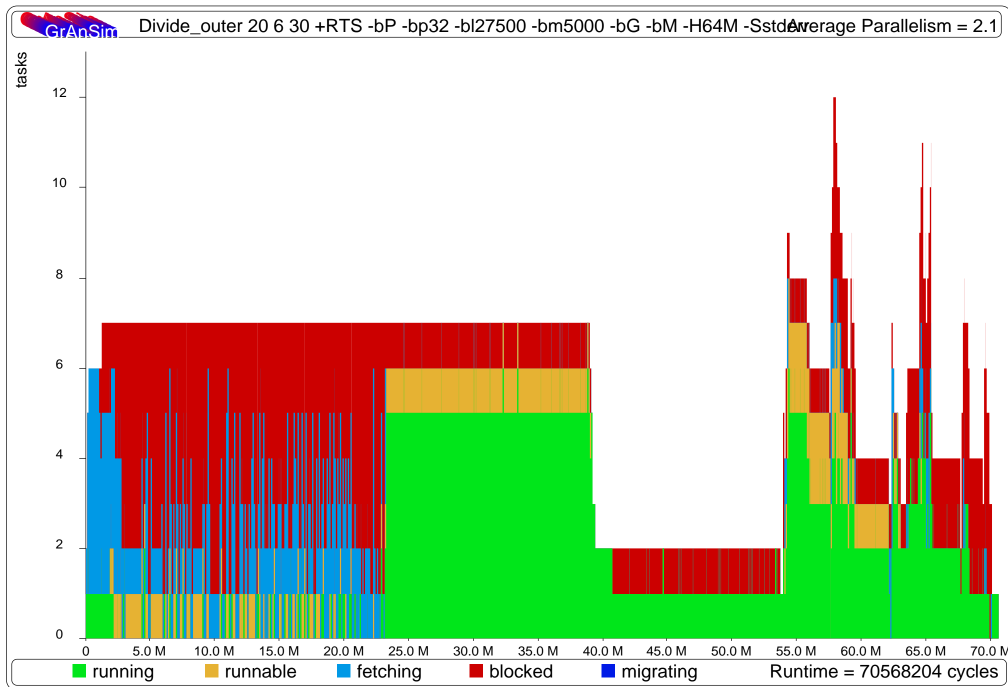


Figure 32: The Activity Profile for Idealised vs Simulated Beowulf (Version III, IV, and V)



Beowulf

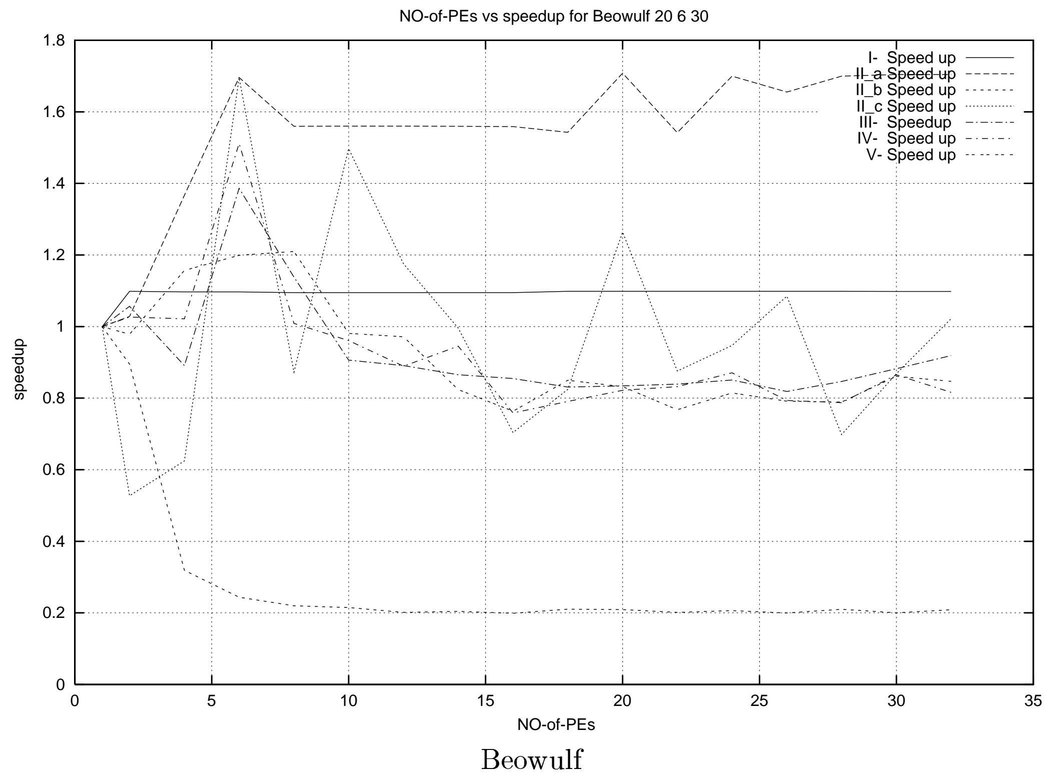


Sun SMP

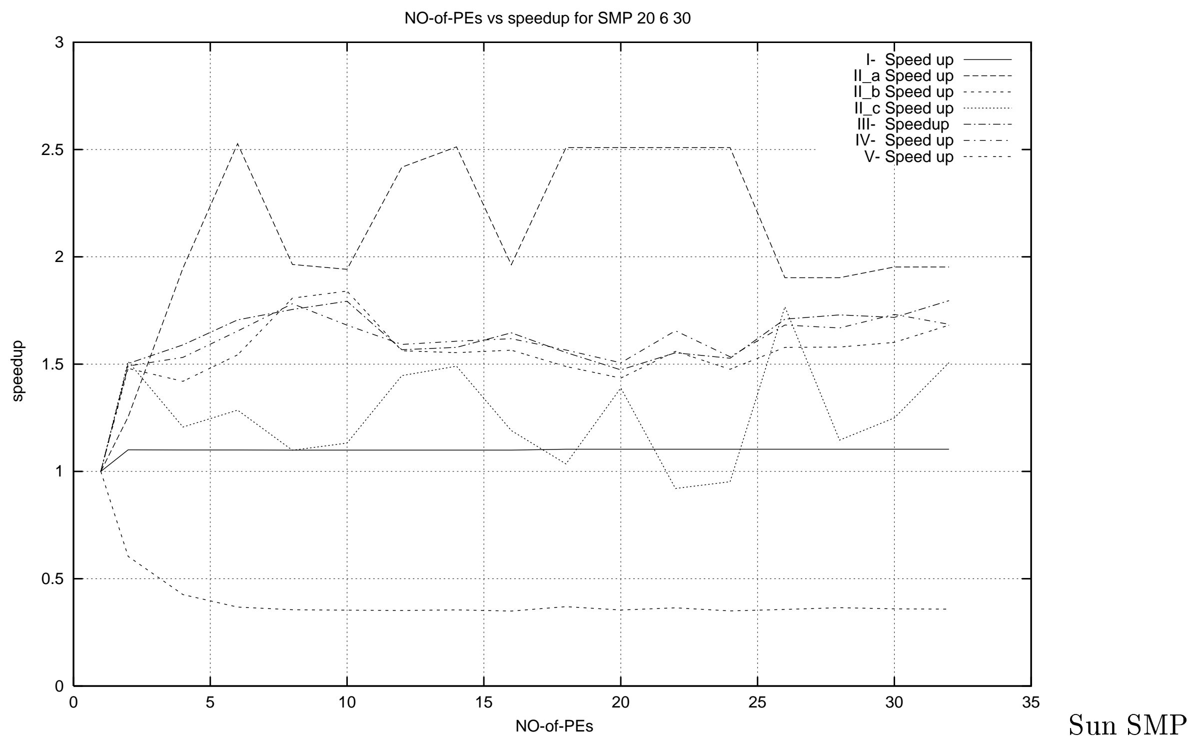
Figure 33: Activity Profile of Beowulf and Sun for Version IIa

3. The comparison in Figure 35 illustrates the chunk size vs the speed up. Both architectures have a similar shape of graph. The best chunk size is 25 or 30 for most versions of the program on both architectures.

4. Figure 33 shows the activity profile from the best version of both architectures. The graphs reflect similar activities except that the Beowulf cluster has more fetching threads because of the higher latency.

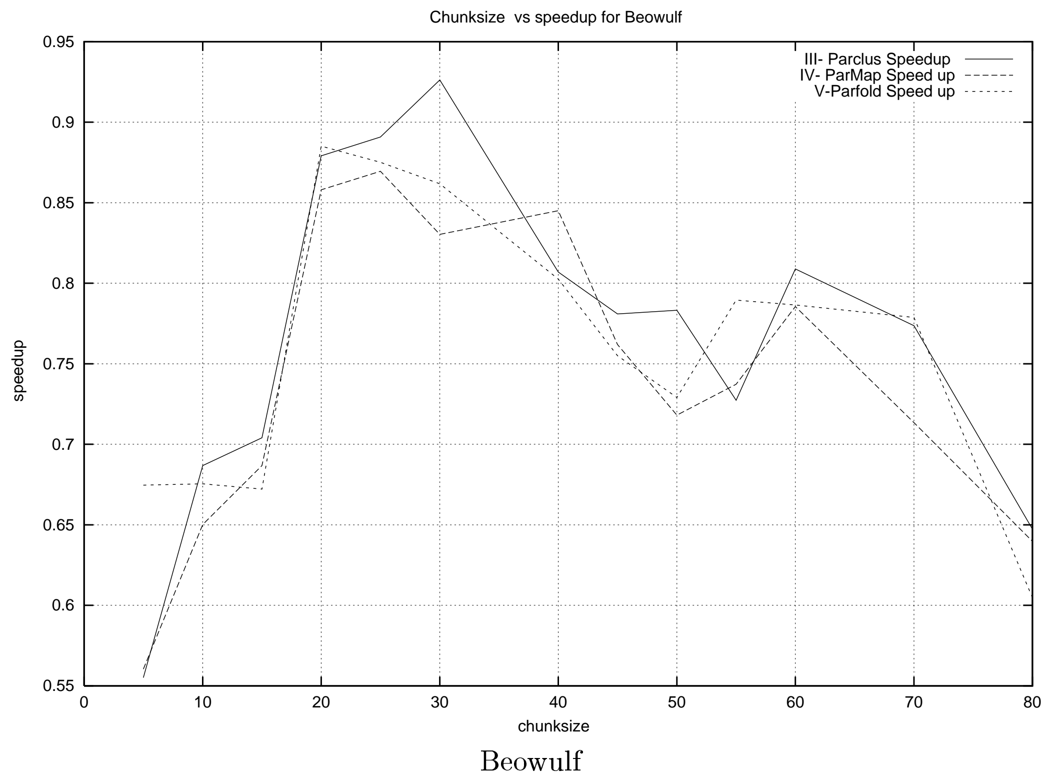


Beowulf

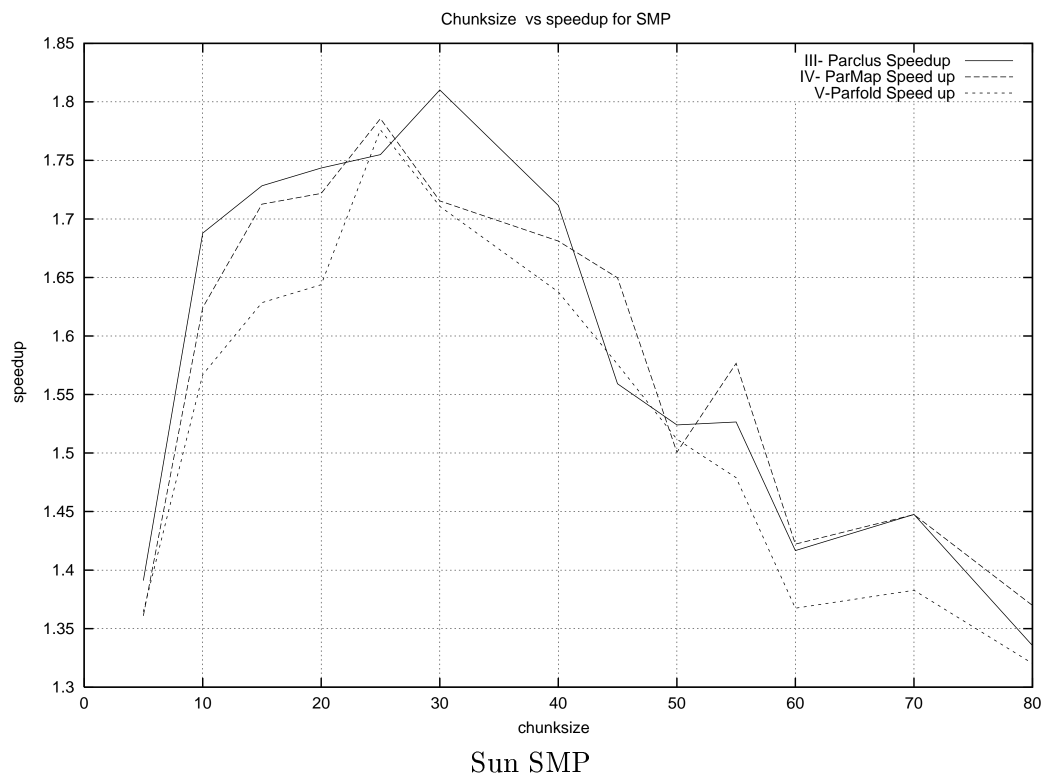


Sun SMP

Figure 34: Speedup vs Numbers Of PES (Simulated Beowulf & SMP)



Beowulf



Sun SMP

Figure 35: Chunk Size vs Speed up for Both Beowulf and SMP Architectures

Chapter 7

Performance Measurements on Two Architectures

The last stage of the methodology as described earlier in chapter 3 section 3.2.5, is to execute the optimised parallel program on real parallel architecture. This chapter will illustrate measurement.

The measurements have been performed on two parallel architectures: the distributed-memory machine (Beowulf cluster), and share memory machine (Sun SPARC Server) which described earlier in section 2.1.1.

The simulated Beowulf and Sun SMP results reported in Tables 3 and 4 predict that version IIa gives the best speedup for both architectures. However, to explore the differences between executing the different versions on a simulated machine and a real machine, it is necessary to test all versions on real machines. The program was measured on two real architectures which are faster than the simulation machine; therefore the input sizes are much bigger than the simulated input sizes.

Program	Speedup	Total Runtime (second)	Generated Tasks	Avg. Tasks Long. (ms)
seq	1	27.7	-	-
I	1.09	27.2	3	.025
IIa	1.9	15.0	21	0.003
IIb	0.8	36.4	861	0.001
IIc	1.4	18.6	16157	0.005
III	1.3	20.9	601	0.007
IV	1.5	18.4	601	0.004
V	1.8	15.5	4226	0.001

Table 5: Real Beowulf Input: 20 40 30 on 4-processor

Program	Speedup	Total Runtime (second)	Generated Tasks	Avg. Tasks Long. (ms)
seq	1	99.9	-	-
I	0.8	123.1	171	0.119
IIa	7.5	13.2	941	0.011
IIb	0.2	140.5	1891	0.004
IIc	1.0	94.0	37821	0.021
III	0.9	107.6	5096	0.057
IV	0.6	155.8	1281	0.013
V	1.0	99.8	7716	0.074

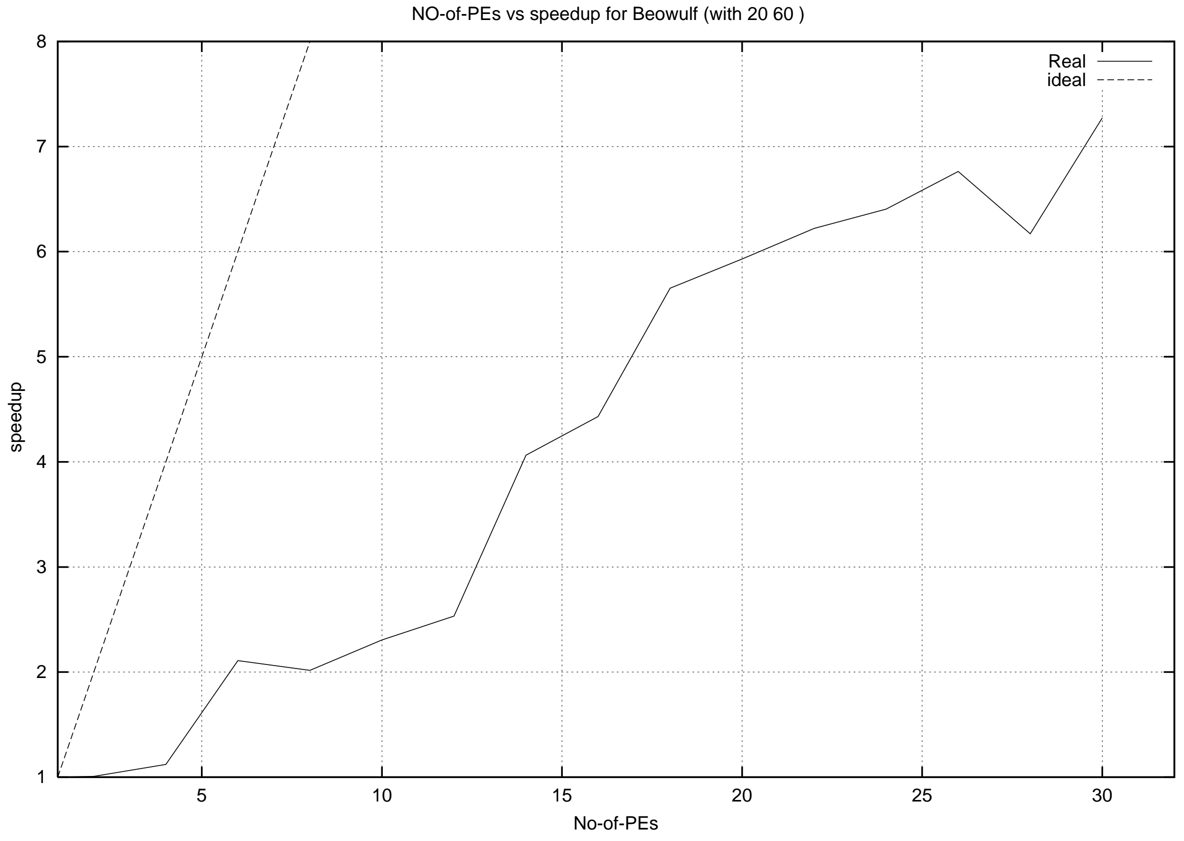
Table 6: Real Beowulf Input: 20 60 30 on 30-processor

7.1 Real Measurement on Beowulf machine

The measurements reported in Tables 5 and 6 show that version IIa gives the best speedup (7.5) on 30-processor. Table 6 summarises the results obtained from the different versions when the program was executed on 30 processors of Beowulf cluster. Table 5 was produced in order to compare Sun SMP results directly with Beowulf results (Table 5). However, results from simulation and real measurement cannot be directly compared, i.e Table 5 with 3, because of the difference in the input size of both measurements.

7.2 Real Measurement on Sun SMP Machine

The measurements reported in Table 7, was for the Sun SMP which limited to 4 processors. The results show that version IIa predicted the best speedup on Sun SMP



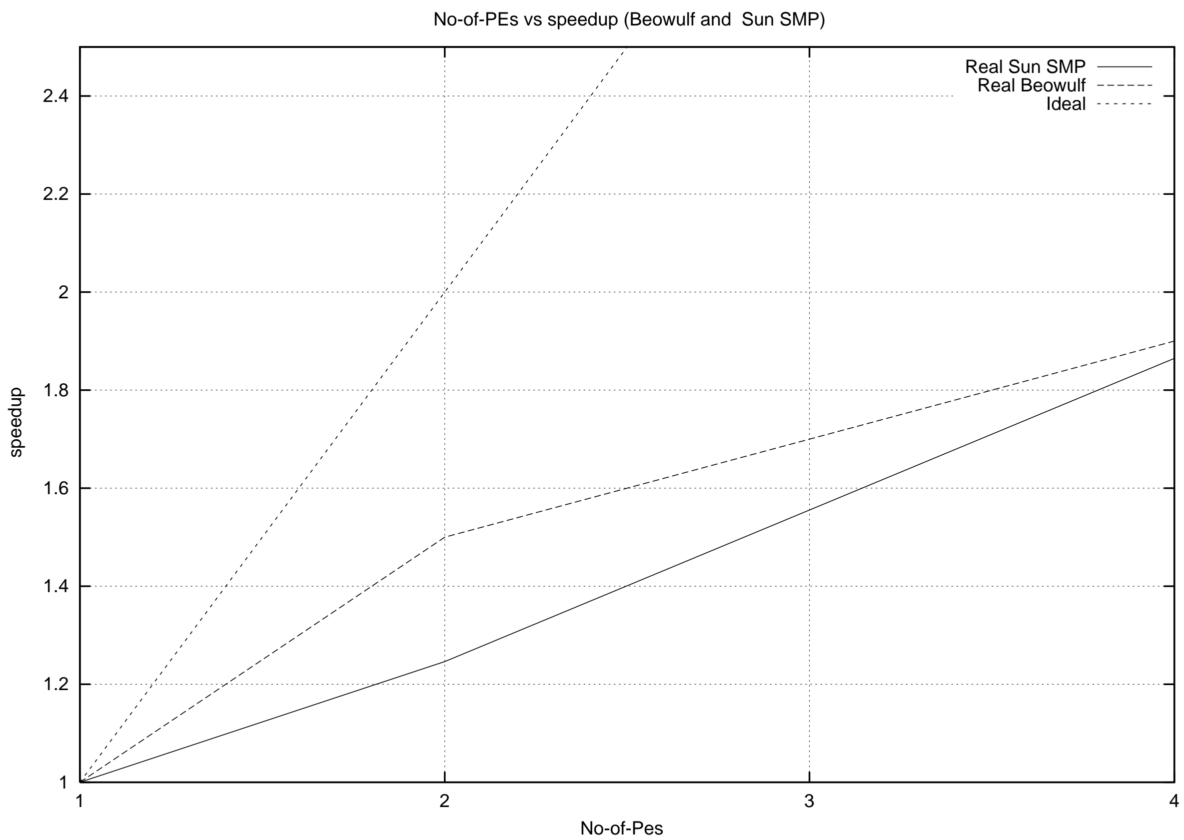


Figure 36: Speedup vs Numbers of PEs Real (Beowulf & SMP)

Program	Speedup	Total Runtime (second)	Generated Tasks	Avg. Tasks Lang. (ms)
seq	1	70.8	-	-
I	0.9	73.8	55	1.34
IIa	1.8	37.9	616	0.061
IIb	0.2	332.0	23940	0.013
IIc	0.4	146.8	12036	0.012
III	0.7	94.8	2585	0.036
IV	0.6	105.8	2850	0.037
V	0.8	87.6	3047	0.029

Table 7: Summary Table of Real Measurement of SMP (20 40) on 4-processors

(1.8). Table 7 summarises the results obtained.

7.3 Discussion of Real Tuning

1. Considering the different versions of the program reported in tables 5, 7, the best version is IIa on both architectures, with speedup 7.5 on the Beowulf and 1.8 on the Sun SMP. This is because version IIa generates big tasks compared with versions IIb to V.The speedup on the Beowulf is better than the idealised speedup (4.2), because of the difference in the input size. The worst version is IIb for both simulation measurements and real measurements, this is owing to the large number of small tasks which increases the amount of communication in the program.

2. Both architectures have approximately similar speedup when executed on four processors, i.e 1.8 on Sun SMP and 1.9 on Beowulf. Figure 36 shows the speedup graphs obtained from Beowulf and SMP.

3. Tables 5 and 7 show that the number of tasks generated by Sun SMP is bigger than the generated tasks by Beowulf for the same input sizes. The reason for this difference is that the Beowulf has bigger latency and higher processor speed, and consequently the idle processor in Beowulf takes more time to fetch tasks. The

GUM mechanism was described in Section 2.8.3). The different number of tasks on both architectures shows that the RTS (Run Time System) can dynamically adjust the granularity of the parallelism to the specific parallel machine.

4. The results from both architectures show that the realistic GranSim simulation accurately predicts which version of the program will give a good performance on real architectures.

7.4 Critique of Multi-Architecture Methodology.

The genetic alignment program exhibits a good performance on both architectures without requiring modification and with the same parameters of the runtime system. The key features of the methodology are as follows:

- The sequential profiling is independent of parallelisation and gives a good sequential program before inserting any parallelism. This is clearly seen from the results obtained: the total execution time of 224 seconds for the initial version dropped to 18.9 seconds for the final sequential version.

- The GranSim simulator provides considerable flexibility to emulate different architectures, including the idealised machine which gives a good indicator of the maximum parallelism that can be obtained. If only a small amount of parallelism is obtained on the idealised simulation then very little is possible on any architecture.

- The idealised version can be reused when targeting new architecture. This saves a programmer from redeveloping his program from scratch when targeting new architecture.

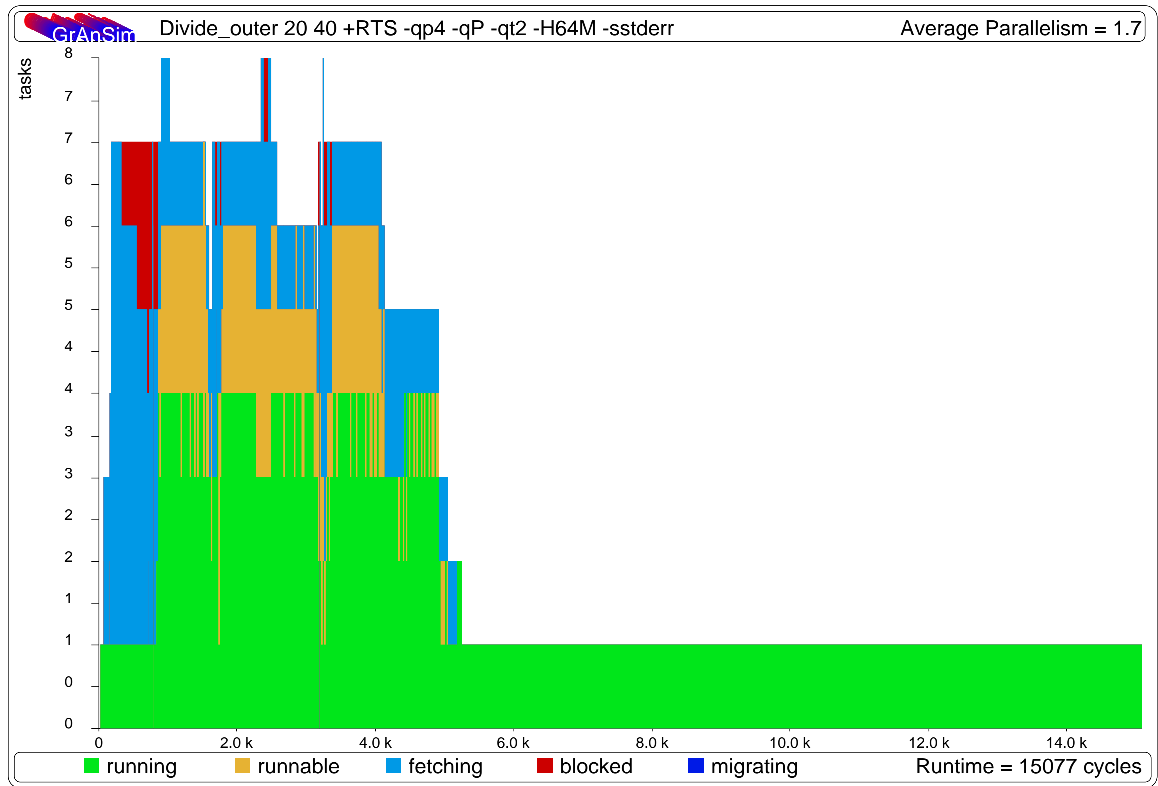
- The idealised simulation results in Table 2 show that the increase in the speedup is related to the increase in task generation Therefore, if a good speedup is needed it is necessary to generate tasks as much as possible, but the real measurement and the realistic measurements indicate that the number of generated tasks depends on the system latency. As can seen from Tables 5 and 7 the number of generated tasks in Beowulf is much smaller than for the Sun SMP.

- Realistic GranSim simulation correctly predicts the program versions that will deliver a good speedup on both architectures. However, there are differences in the shape of the activity profiles produced from GranSim and GUM, as shown in figures 33 and 37. Unfortunately it is not possible to compare the figures directly because of the difference in input sizes. Some of the differences are the result of system issues; e.g. in GUM, it is possible to control the number of tasks created on PE while this is not possible under GranSim. Moreover, GranSim does not cover the communication behaviour of the machine: the bandwidth of the communication channel and the topology of the underlying machine. GranSim assumes that the latency between two processors is independent of the communication traffic [58].

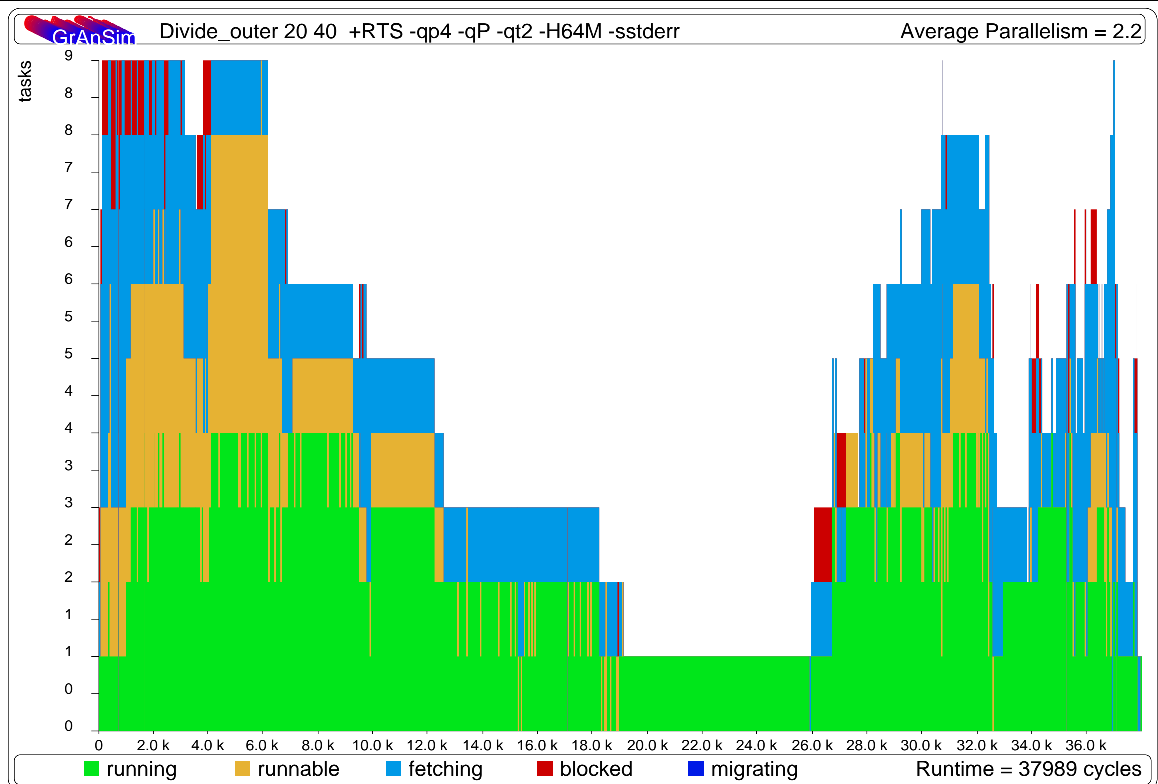
- No changes are required to the program source, to move the genetic parallel program from Beowulf cluster to Sun SMP architecture. This is owing to the fact that the programmer controls only a few parallel aspects, as most aspects are controlled by the runtime system, such as thread creation, communication between tasks, and task placement. The best performance on both architectures is obtained from the same parallel version of the program (version IIa).

The model described here supports the claims that the high level coordination in parallel functional languages facilitates software development for multiple architectures, by showing that minimal program changes are needed to move an application written in GpH from one architecture to another.

All chapters from Chapter 4 and including this chapter describe the implementation of the proposed methodology. The implementation shows that it is necessary to control the generated tasks from GpH program by considering the underlying parameters. The next chapter will discuss new architecture independent functions and their implementation.



Beowulf



Sun SMP

Figure 37: Actual profiles of version IIa of Real Beowulf & SMP, Input 20 60

Chapter 8

Enhancement of Architecture Independence in GpH

8.1 Overview

From the experiment described throughout the previous chapters and others [5, 46], it may be seen that many GpH programs often create a massive amount of parallelism. The philosophy of GUM's load balance mechanism is to allow a high amount of potential parallelism and distribute the potential work in the form of sparks. In addition to that, it provides a kind of management to make spark creation cheap, thus minimising the cost of spark movement between processors. Also any spark turned into a thread is evaluated by its PE [4].

This mechanism works well if the GpH program generates big potential tasks, i.e. large tasks granularity. As seen from the genetic alignment program the best performance is obtained from the version with large parallel tasks. Therefore, the author proposes to build a new model for GpH that uses information about the underlying architecture when it generates potential parallel tasks. The architectural parameters

that may be involved are the number of available processors, system latency and the processor clock speed. Figure 38 shows the new GpH model.

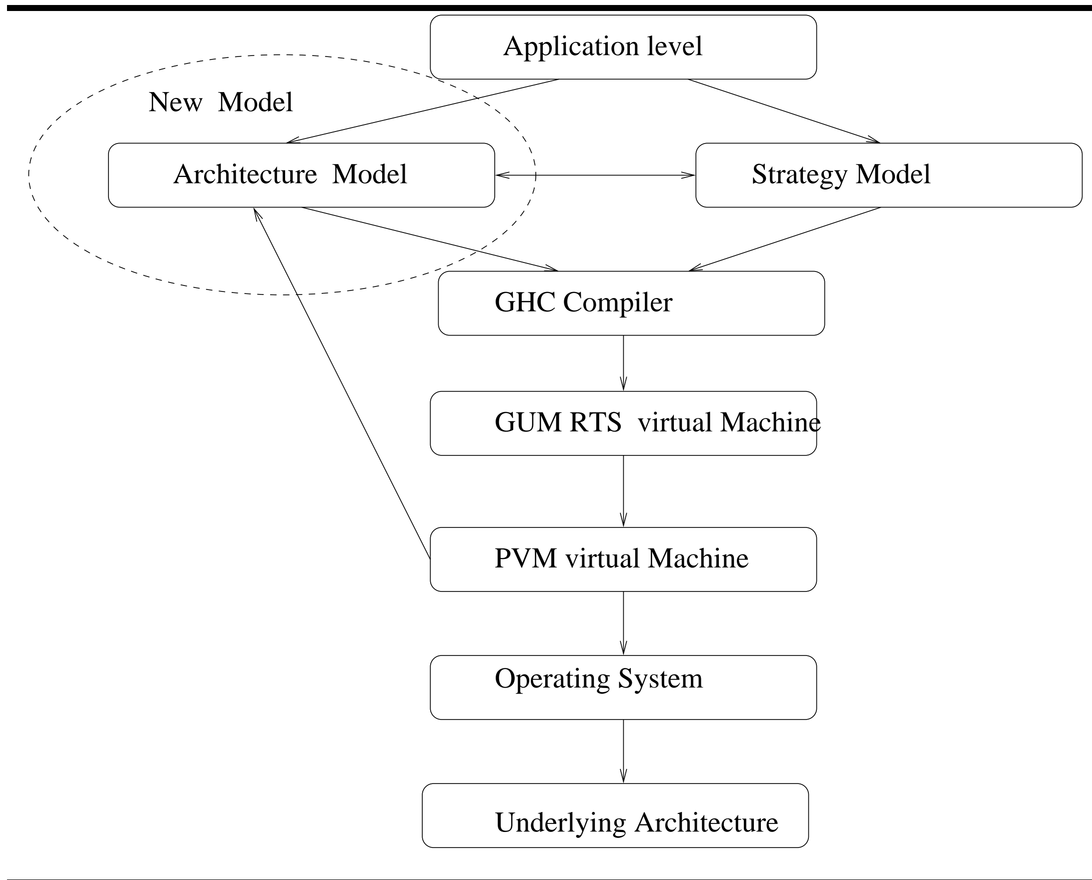


Figure 38: A New GpH Structure

8.2 Extracting Architecture Characteristics

The most important architecture parameter to be abstracted in this section is the number of processors. The motivation for abstracting the number of processors (PEs) is that the most important characteristic for parallel program is the granularity. The typical techniques for parallelisation such as `parMap` over long lists generate much more parallelism than is needed. Therefore, it is important to achieve a good granularity even with changing the numbers of PEs without source code change. In other words the implementation of many parallel functions, e.g. a `parMap` function, often yields very

fine task granularity. Even if clustering is used the task granularity still depends on the programmer estimation and it does not change dynamically. As another example, the divide and conquer sometimes ends up with many small parallel tasks that reduce performance; the programmer often controls this situation by threshold. The number of processors (PEs parameter) can be used to write strategic functions which minimise the number of generated tasks, and can be used to define new strategies. Figure 39 shows the function which returns the number of processors(PEs). Sections 8.3 and 8.4 will show the implementation of the PEs parameter in the genetic alignment program.

```
/* This C  program  function return the actual PEs runing by PVM */
#include <stdio.h>
#include "/net/dazdak/fp/pvm3/include/pvm3.h"

int numberPEs(void);

int numberPEs(void)
{
   struct pvmhostinfo *hostp;
   int  nhost, narch;
    /* get configuration of the parallel machine */
    pvm_config( &nhost, &narch, &hostp )
    return nhost ;
}
```

Figure 39: A Number of PEs Function

8.3 Generic Architecture Adapting Strategies

The new strategic functions can be added to the strategy library used with the GpH system as shown in Figure 38. The new strategies use information about underlying

```
module Architecture( pe,parMapPe, chunksize )
          where
import Strategies


cluster ::  Int -> [a] -> [[a]]
cluster n [] = []
cluster n xs = take n xs :  cluster n (drop n xs)



pe = pes ()
pes ::() -> Int
pes null = unsafePerformIO (_ccall_ numberPEs)



parMapPe::(NFData a,NFData b)⇒ Strategy [b]→Int→(a → b) → [a] → [b]
parMapPe strat pp f [] = []
parMapPe strat pp f xs = clisst
          where
               clist = cluster num xs
               nn = length xs
               num = if (nn `div` pp ) == 0 then 1 else (nn `div` pp)
               plist =map (map f) clist `using` parList strat
               clisst = concat plist
```

Figure 40: The New Functions already Built in Architecture Model

architecture to minimise the number of tasks generated by a GpH program. For example a new strategies `parMapPe` was defined to chunk the given list automatically, based on the number of processors (PEs). The `parMapPe` function guarantees that the number of generated tasks is equal to the number of processors. The same technique could be used to defined a new `parfoldr` strategy which splits the given list into a sublist and folds the function over them in parallel. The code of `parMapPe` is shown in Figure 40. Also a general divide conquer function can be defined, the number of processors can be used to determine the maximum tree level as shown in Figure 41.

```
seqdiv ::  (a → Bool) →(a→ b) → (a→ [a])
           → ( [b]→ b) → a→ b
seqdiv trivial solve split combine x
           | trivial x = solve x
           | otherwise = combine child
               where
               child = map (seqdiv trivial solve split combine ) (split x)
-- Parallel divide function
pardiv ::  Int → (a → Bool) →(a→ b) → (a→ [a])
           → ( [b]→ b) → a→ b
pardiv 0 trivial solve split combine x = seqdiv trivial solve split combine x
pardiv (pes-1) trivial solve split combine x
           | trivial x = solve x
           | otherwise = combine child
               where
               child = parMap rnf (pardiv (pes-1) trivial solve split combine )
          (split x)
```

Figure 41: The New General Divide Conquer Function

Beowulf Implementation. The implementation of the new function on the Beowulf cluster shows some improvement in speedup when there are more than twelve processors and no change when there are fewer processors. The function was tested on version III of the genetic alignment program. In the genetic alignment program the `Form_pin` is applied to a big list; e.g. when the input is 20 sequences of length of 60, the length of the subsequences list applied to the function is 2015 elements. This generates 68 parallel tasks for each recursive call if the chunk size is set to 30. Of course fewer tasks are generated for each recursive call, but still version III generates a huge number of parallel tasks. The total number of generated tasks from version III as reported in table 6 is 5096 tasks. In contrast the new`parMapPe` will generate fewer parallel tasks depending on the number of processors; e.g. if the `parMapPe` function is called with the same list and 4 processors, it will generate only four parallel tasks for each recursive call. It generates 771 tasks in total from the same input size. Figure 42 illustrates the

improved performance of the new function.

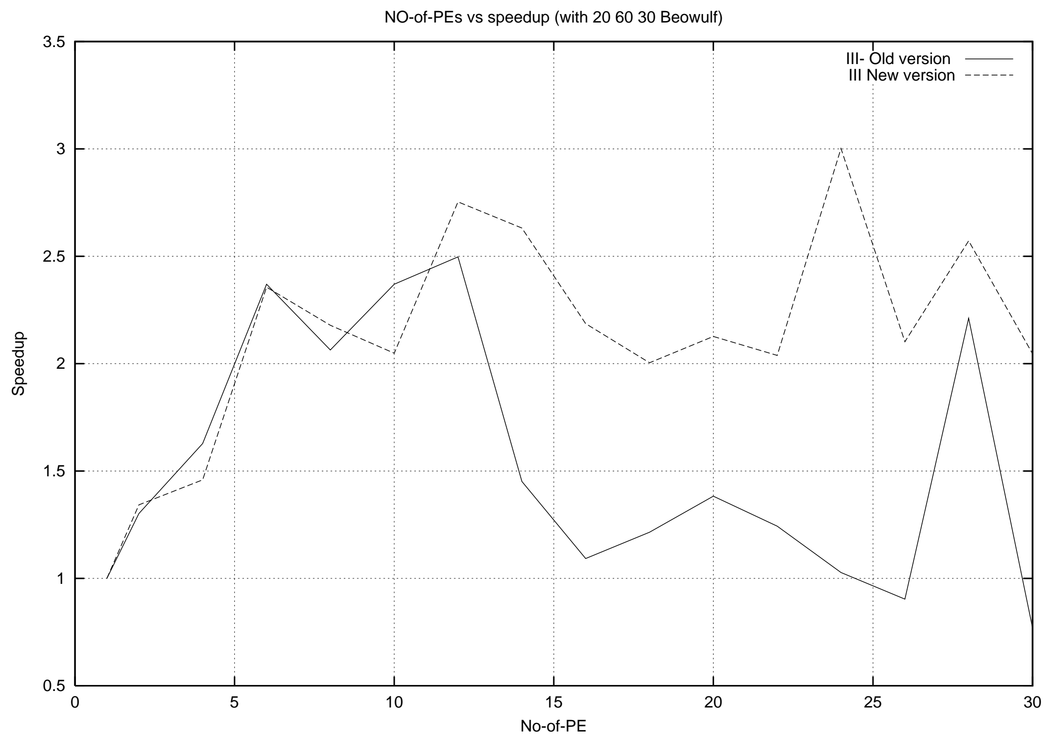


Figure 42: The New parMapPe Relative Speedup for Beowulf

Sun SMP Implementation. The `parMapPe` function was also tested using version III of the genetic alignment program. The experiment shows some improvement in the speedup; e.g. on four processors the speedup increased from 0.8 to 1.1. Figure 43 shows both the speedup of the old clustering version (III) and the speed up of an automatic clustering version.

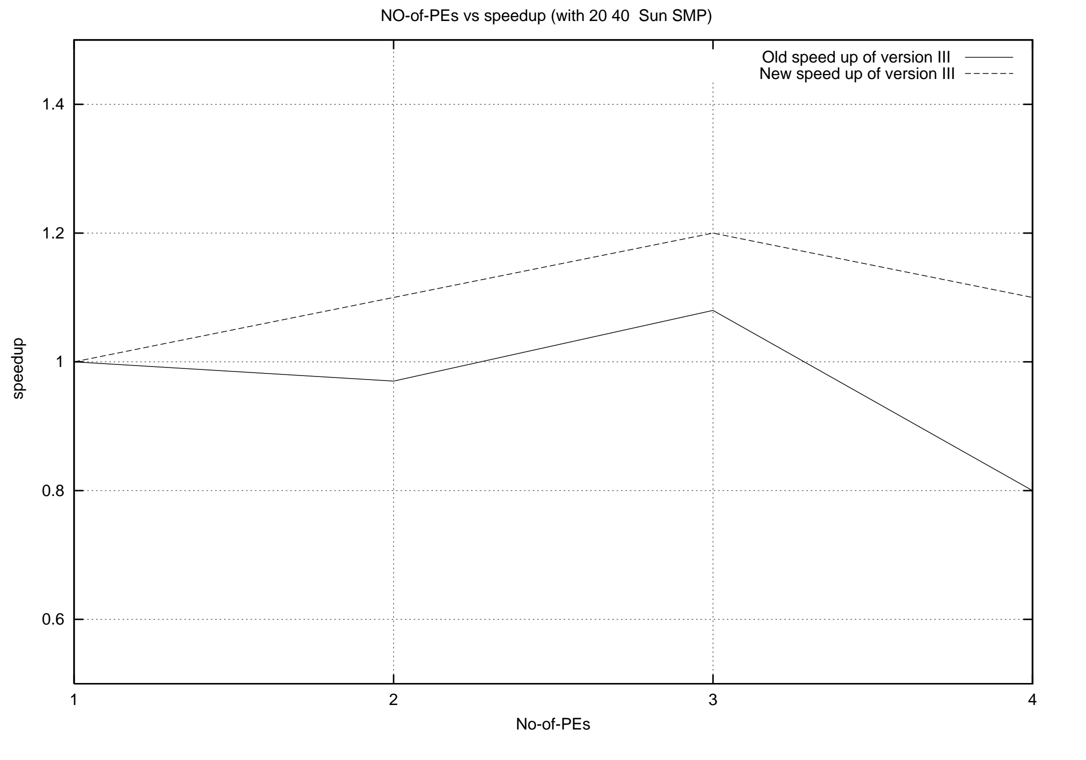


Figure 43: The New parMapPe Relative speedup for SunSMP (Input 20 40)

8.4 Architecture Adapting Strategies for Specific Application

The parallelism in divide and conquer comes from the fact that a given task is split into sub-tasks that can be evaluated in parallel. This technique is used in the genetic alignment program. The `divide` function generates three parallel tasks for each recursive call. It is possible to control the generated tasks from the function by passing a new parameter, as shown in Figure 44. This parameter is used to limit the depth of parallelism generated in the divide function call tree. The divide function generates three parallel tasks for each recursive call. To match the number of tasks with the number of PEs, the new `pes` parameter passed to the initial call is computed as log_3`(pe)`. If the length of the given list is smaller than the number of processors, the result from

```
Align_chunk ::  Int→ [Sequence] → [Sequence]
Align_chunk pes [] = []
Align_chunk pes xs = fun_align all_res
           where
                best = Bestpin xs -- Find the best pin from xs
                all_res = divide pes xs best -- Split and align the xs
divide ::  Int → [Sequence] → Pin → [Sequence]
divide pes [] [] = []
divide pes xs [] = xs   -- Basic alignment
divide pes xs pin = (combine pin res_left res_right res_unpinch )
      `demanding` strategy
           where
               (rightch,leftch,unpinch1) = splitting_sequences pin xs
               unpinch = lead_function pin unpinch1
               res_unpinch = align_chunk (pes-1) unpinch
               res_right = align_chunk (pes-1) rightch
               res_left = align_chunk (pes-1) leftch

               strategy =if pes <= 0 then () else ( rnf res_left `par`
                     rnf res_right `par`
                     rnf res_unpinch)
```

Figure 44: A New Divide Function

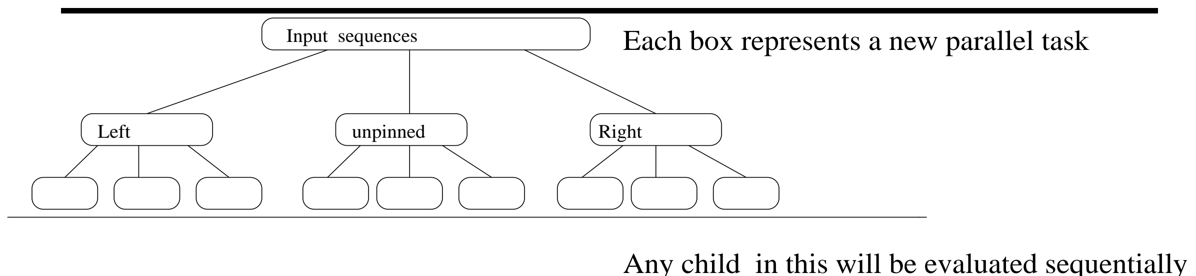


Figure 45: Divide Function Diagram when it called by 2 PEs

log_3`(pe)` is zero, and the chunk size in this case is set to one. For example, if the divide function is called with 1 processor it will evaluate only the first two levels in parallel; the rest of the tree will be evaluated sequentially, as shown in Figure 45.

Beowulf Implementation. The implementation of the new function improves the utilisation of the system resources. The utilisation of processors was increased as shown in Figure 46, when the program was executed on the Beowulf cluster. Also the average parallelism and runtime were improved by factor, e.g from 4.0 to 4.9 and from 20.2s to 15.0s respectively. Moreover, the implementation shows good improvement in the speedup, as shown in Figure 47.

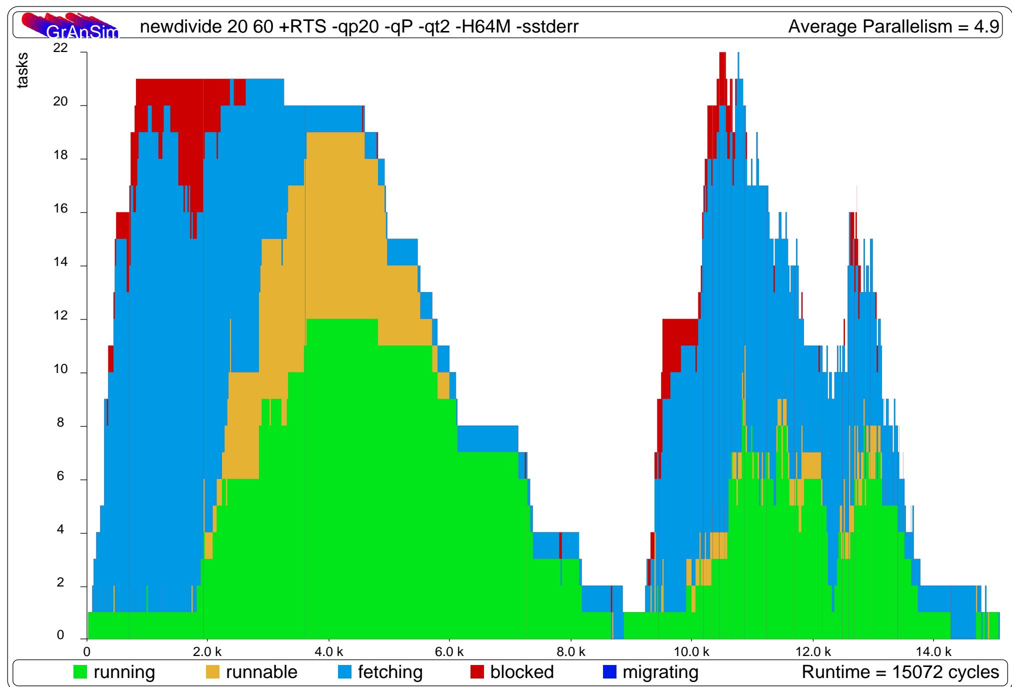
Sun SMP Implementation. Figure 48 shows the improvement in the speed up when the number of processors is used in version IIa. The speedup was increased; e.g. on four processors the speedup was increased from 1.8 to 2.4.

8.5 Summary

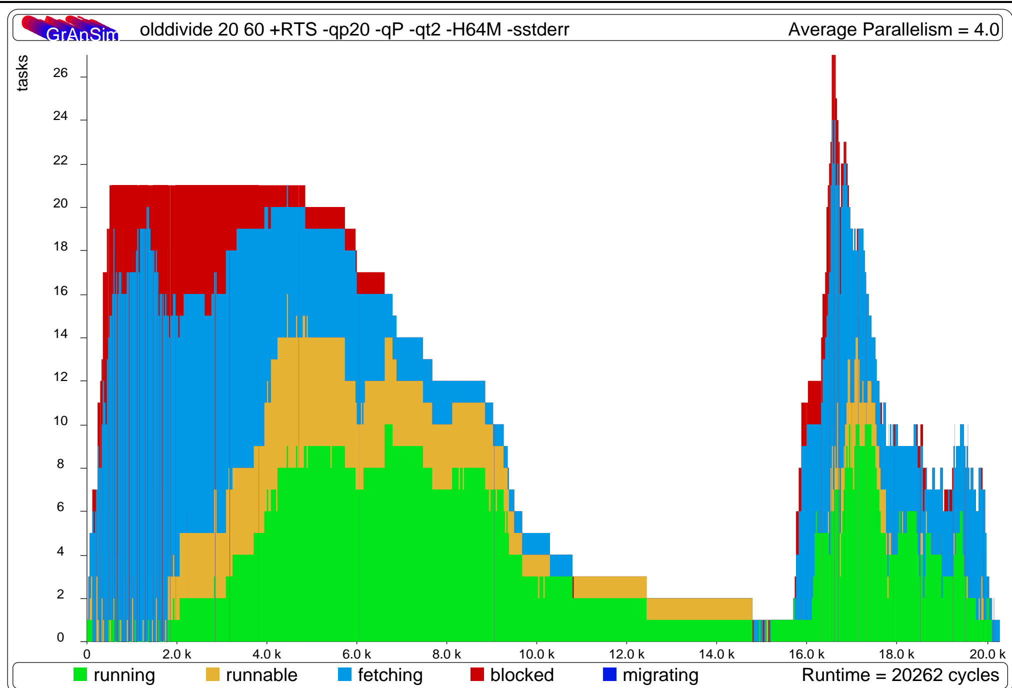
This chapter has shown that the utilising the key architectural parameters in GpH programs give better performance on both architectures. The key architectural parameters have two levels of implementation: the first level is at the standard strategy library in the GpH, where the architectural parameters are used to define new generic strategies and the parameters are hidden from the programmer. In the second level the key architectural parameters can be used by the programmer to tune performance, e.g. adopting the task granularity of the GpH program. The implementation shows improvement in the speedup of version III when the `parMapPe` function is applied (see Figures 42 and 43), because the `parMapPe` generates fewer tasks than `parMap`. The

generation of the parallel tasks is dynamically controlled by the number of processors and the list length. Moreover, Figure 46 shows the improvement in the system resource utilisation of version IIa; e.g the total execution time dropped from 20.2s to 15.0s when the program is executed on 20 PEs. This improvement is owing to the fact that only three levels of parallelism are generated by the divide function.

The implementation of the improved strategies on the genetic alignment program which uses both classes of data-parallelism and divide and conquer parallelism gave better performance. Therefore, using the modified strategies on other applications of these classes should improve performance in the same way that refined Skeletons [19] can improve the performance of an entire class of applications. This shows that the programming techniques discussed here are relevant in a broader context.



(New)



(Old)

Figure 46: Activity Profile for New and Old Divide Function (20-Processors)

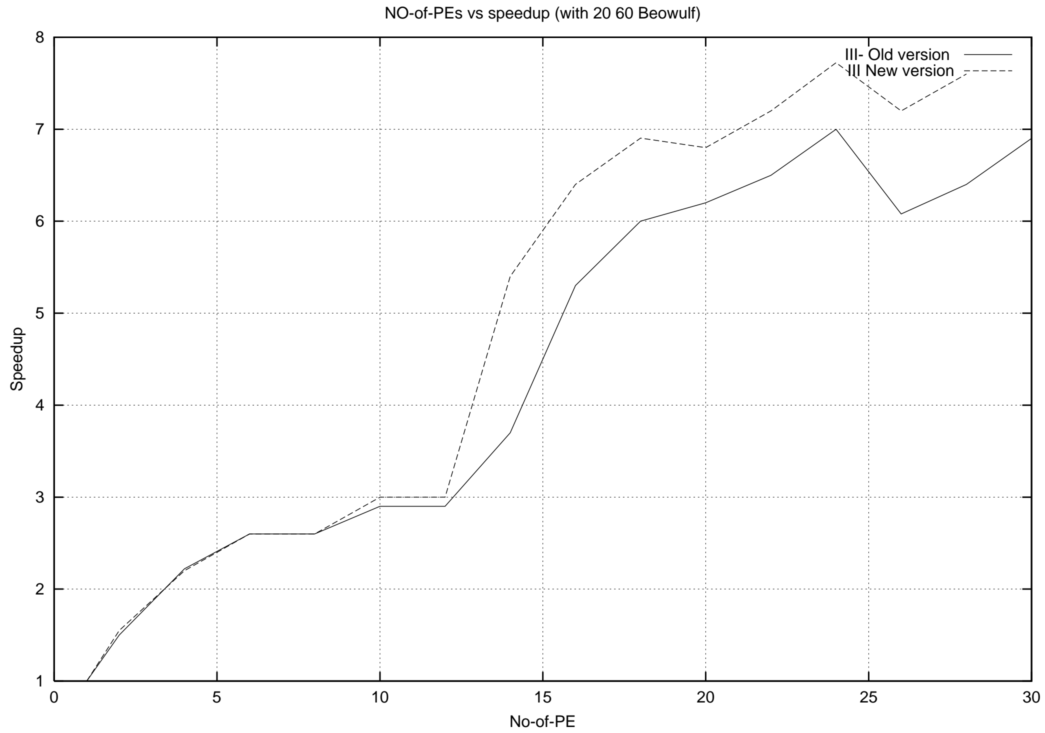


Figure 47: The New Divide Function Relative speedup for Beowulf

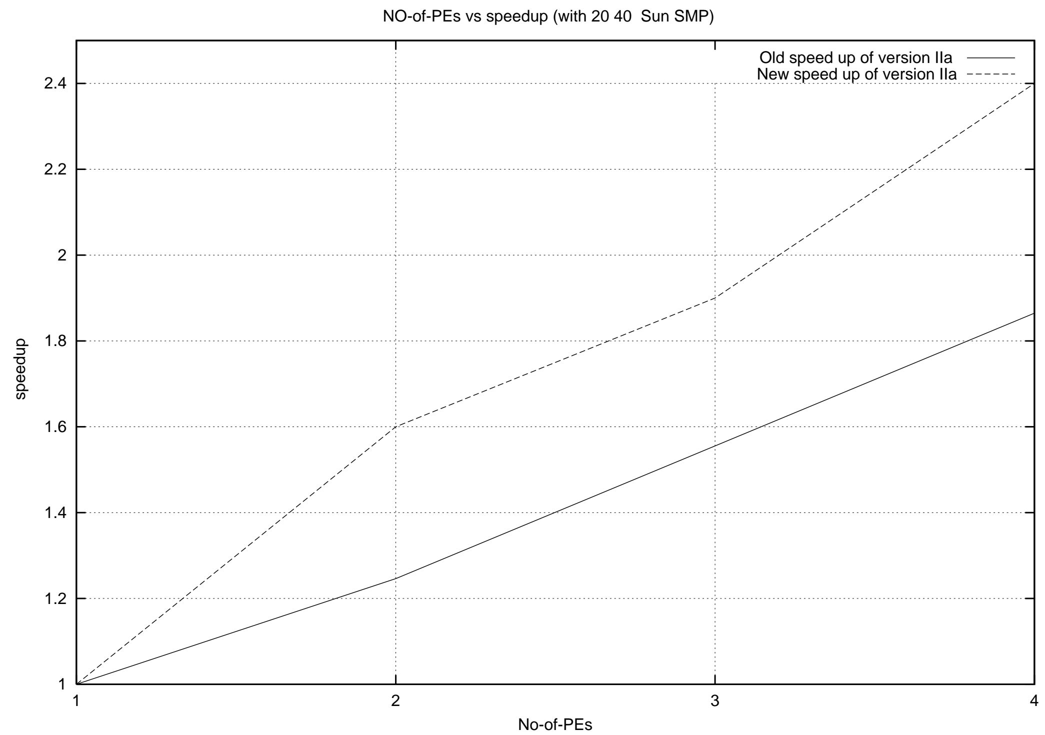


Figure 48: The New Divide Function Relative speedup for Sun SMP

Chapter 9

Conclusions

9.1 Introduction

One means of supporting architecture independent parallel programming is to use programming model with a high level coordination. The model should achieve good performance across a wide range of parallel architectures. It should hide most of the parallel aspects from the programmer. Also, it should be easy to deal with programs of very different structure.

This thesis has investigated the use of high level functional languages for architecture independent parallel programming. The proposed methodology for GpH [46, 5] has been used to develop a substantial application and also extended (see Figure 5 in Chapter 3). The application has been developed for two different architectures in Chapters 4, 5, 6, 7.

9.2 Achievements

9.2.1 Assessing a Multi-Architecture Parallel Programming Methodology

The first systemic evaluation of Trinder and Loidl's multi architecture programming methodology for GpH has been reported. It was explored using a genetic alignment program, which was developed using the methodology for the first time. The implementation has shown the importance of each stage in the methodology. Sequential optimisation gives a massive improvement in the execution time and memory consumption (see Section 4.4.4). It has been shown that a realistic GranSim simulation correctly predicts the program version that gives the best performance on both real architectures. Also, the results show that the methodology produces a program with acceptable performance on both architectures, and this supports the conclusion that high level functional programming is a good approach to architecture independent parallel programming.

The methodology has the following limitation. Firstly GranSim is too slow and the profiles consume considerable disk space. As result it is not possible to run the experiment with the same input size as used in real architectures. This problem is shared with other approaches using simulators. Secondly GranSim does not include all features of the parallel architecture, i.e there is no model of communications bandwidth between PEs. Thirdly the idealised stage may lead to the generation of parallel programs that do not deliver good performance on real architecture, e.g versions IIb, IIc, III, IV, V above.

9.2.2 Extended The Architecture Independent Capabilities of GpH

To improve the architecture independence of GpH new parallel coordination constructs have been designed, implemented and measured. The primitives extract key architecture specific properties of the machine and use them to control coordination, often without exposing the properties to the programmer. Improved parallel performance is demonstrated using the primitives. Figure 38 in Chapter 8 shows the new structure of the extended methodology for GpH. As seen from the figure the enhancement were made on the GpH structure rather than the methodology stage shown figure 5 in Chapter 3.

9.3 Limitations

The work has the following limitations.

- The methodology has been investigated using one application. To have a stronger basis for conclusions on the usefulness of the extended methodology it would be good if more programs were developed using the same methodology.
- The genetic alignment program was tested on only two available architectures. It would be useful if the program could be tested on more architectures. This is very good evidence to support the proposed methodology.
- Because of time limitations, only one parameter was abstracted and implemented in the extended methodology. It would be useful to abstract more parameters from the underlying architecture, such as latency and packing cost.
- The real and simulation results are not directly comparable for several reasons, including the difference in the input size, and the difference in the runtime options

between the GranSim and GUM runtime system, as mentioned in the first point.

9.4 Future Work

To have strong support to the methodology, it would be useful for it to be used in developing more applications in the area of computer architecture. In addition, there is scope for it to be tested on more parallel architectures.

It would be useful to a cost model technique to improve the architecture independence in the GpH mode. It would be desirable to investigate the use of the sequential profile information to spark the parallel tasks.

The extended methodology in Chapter 8 could be enhanced further by examining other machine characteristics such as latency and processor speed. In particular, the information could be used in automatically determining the chunk size in data parallel programs.

Appendix A

Source Code for The Genetic Alignment Program

The appendices are organised as follows: Appendix A.1 contains the code for optimised sequential version, Appendix A.2 contains the code for the best parallel version (IIa) which delivers best speedup on both architectures. Appendix A.3 contains modified functions for a finite-Map implementation see Section 4.4.3 for more details.

A.1 Final Sequential Version

This section contains the final code of optimised sequential version (V).

```
 module Main where
import System(getArgs)
import List
import Random
data Aminoacid = A | C | U | G | D |I
  deriving (Read,Show,Eq,Ord)

type Sequence1 = [Aminoacid]
type Sequence =  [Int]
type Pin = [Int]
type SubSequence = [Int]

align_chunk ::  [Sequence] -> -- List of input sequences.
                [Sequence]    -- Aligned sequences.
align_chunk [] = []
align_chunk xs  = fun_align all_res
     where
```

```
      best = Bestpin xs
      all_res = divide   xs best

divide ::   [Sequence] -> -- List of input sequences
            Pin -> -- Best pin
    [Sequence]   -- List of aligned sequences.
divide  [] [] = []
divide  xs [] = xs -- Basic alignment
divide  xs pin = (combine pin res_lift
                              res_right
                              res_unpinch )

        where
        (rightch,leftch,unpinch1) = splitting_sequences pin xs
        unpinch = lead_function pin unpinch1
        res_unpinch = align_chunk  unpinch
        res_right = align_chunk   rightch
        res_lift = align_chunk   leftch

combine :: Pin -> [Sequence] ->[Sequence] ->[Sequence] -> [Sequence]
combine  pin left_seqs right_seqs unpinned_seqs
         = ( zipWith ( cat_sequence pin) left_seqs right_seqs)
            ++ unpinned_seqs
         where
           cat_sequence :: Sequence -> Sequence ->
                           Sequence -> Sequence
   cat_sequence pin ls rs = ls ++ pin ++ rs


Bestpin ::  [Sequence] -> -- List of input sequences.
                       Pin    -- Best pin as output.
Bestpin []= []
Bestpin xs =  best_pin   pins_dis
  where
    all_substring  = substring_sequences xs xs
    pins = map  fst( extract_max_pins all_substring )
    extract_longest_pins = longest_pin  pins
    pins_dis =  pin_average_distance   extract_longest_pins  xs


---------------------------------------------------------------
-- ALL FUNCTIONS CALLING BY Bestpin FUNCTION       --
---------------------------------------------------------------
substring_sequences :: [ Sequence] ->
                       [ Sequence] ->      -- The input sequences
                       [(Sequence,Int )]   -- List of substring list
substring_sequences [] [] = []
substring_sequences [] ys = []
substring_sequences (x:xs) ys  =  nub res1
          where
   res =   subseq x
   res1= form_pins res  ys ++ substring_sequences xs   ys

subseq :: Sequence    ->
          [SubSequence ]
subseq (x:xs) = inits (x:xs)++  subseq xs
subseq [] = []

form_pins :: [SubSequence] ->
             [Sequence ]   ->
```

```
     [(Pin,Int )]
form_pins [][] = []
form_pins[] ys = []
form_pins(x:xs) ys
   | num > 1 =  (x,num): form_pins xs ys
   | otherwise = from_pins xs ys
   where
   num = from_pins' x ys

from_pins' :: SubSequence    ->  -- Single element from substring list
              [Sequence]  ->  -- A list of input sequences
       Int            -- Number of occurrences as pins
from_pins' [] [] = 0
from_pins' m []  = 0
from_pins' m (x:xs)
  |((check_for_snd_appears == Nothing )
    &&(check_for_appears /= Nothing )) = 1+ form_pins' m xs
  |otherwise =  form_pins' m xs
  where
   check_for_appears   = locate_pin m x
   where_pin_appears = pin_value(check_for_appears)
   reset_of_sequence  = drop (where_pin_appears + length m ) x
   check_for_snd_appears =  locate_pin  m reset_of_sequence

extract_max_pins :: [(Pin,Int )] -> [ (Pin ,Int )]
extract_max_pins [] =[]
extract_max_pins ((p,n):xss) = foldr (extract_max_pins')  [(p,n)]  xss


extract_max_pins' :: (Pin ,Int)  ->[ (Pin,Int)] ->  [(Pin,Int)]
extract_max_pins'  (p,n) aa_pin@((p',n'):_)
     | n' >  n  =   aa_pin
     | n' ==  n  = aa_pin ++ [(p,n)]
     | otherwise = [(p,n)]

longest_pin  ::  [Pin] ->[Pin]
longest_pin [] = []
longest_pin (xs:xss) = foldr  (longest_pin') [xs] xss

longest_pin' ::  Pin  ->[Pin] ->[Pin]
longest_pin' pin  xs@(pin':_)
     | length pin' >  length pin   = xs
     | length pin' ==  length pin  = (pin:xs)
     |  otherwise = [pin]

pin_average_distance :: [Pin] ->
                        [Sequence]  ->
[(Pin, Int ) ]

pin_average_distance  [] [] = []
pin_average_distance [] _ = []
pin_average_distance _ [] = []
pin_average_distance   xss   yss = [ (xs ,((sum (map (
    pin_average_distance'  xs) yss ) `div` length yss )))| xs <- xss]
    where
    pin_average_distance'  :: Pin   -> Sequence ->   Int
    pin_average_distance' [] [] = 0
    pin_average_distance' xs ys
      | check_res == Nothing  = 0
      | otherwise = abs(((length ys) `div` 2) -
```

```
       ((pin_value ( check_res )) + (length xs  `div` 2)))
        where
 check_res = locate_pin xs ys

best_pin ::  [(Pin,Int)] ->
             Sequence
best_pin [] = []
best_pin ((p,n):xs) = fst( best_pin' (p,n)  xs)
best_pin' (p,n) [] = (p,n)
best_pin' (p',n') ((p,n):xs)
  |  n <= n'   =   best_pin' (p,n) xs
  | otherwise = best_pin' (p',n') xs

----------------------------------------------------------------
-- ALL FUNCTIONS CALLING BY divide  FUNCTION                  --
----------------------------------------------------------------
splitting_sequences :: Pin  ->
                    [Sequence] ->
   ([Sequence] , [Sequence] , [Sequence])
splitting_sequences [] [] = ([], [] ,[])
splitting_sequences [] ys = (ys, [], [])
splitting_sequences xs ys =(right_chunk , lift_chunk , unpinned)
       where
        right_chunk = right_sequence xs ys
        lift_chunk = left_sequence xs ys
        unpinned = unpinned_chunk xs ys

right_sequence :: Pin  -> -- best pin.
                 [Sequence] -> -- List of input sequences.
         [Sequence]   -- List of right sequences.
right_sequence [] [] = []
right_sequence [] ys = ys
right_sequence xs ys = map (right_sequence' xs )
                           (pined_chunk xs ys)
    where
    right_sequence' :: Pin ->  Sequence -> Sequence
    right_sequence' [] [] = []
    right_sequence' [] ys = ys
    right_sequence' xs ys
       | (num_pin + length xs)  == length ys  = []
       | check_res /= Nothing = drop (  num_pin + (length  xs ))  ys
       where
       check_res =  locate_pin xs ys
       num_pin  = pin_value( check_res)
left_sequence :: Pin  ->      -- best pin.
                [Sequence] -> -- List of input sequences.
        [Sequence]    -- List of left sequences.
left_sequence [] [] = []
left_sequence [] ys = ys
left_sequence xs ys = map (left_sequence' xs ) (pined_chunk xs ys)
    where
     left_sequence' :: Pin ->  Sequence -> Sequence
     left_sequence' [] [] = []
     left_sequence' [] ys = ys
     left_sequence' xs ys
        | num_pin  > 0 = take ( num_pin ) ys
        | otherwise = []
where
 num_pin =   pin_value (locate_pin xs ys)
```

```
unpinned_chunk    xs ys = filter(\y -> (locate_pin xs y) == Nothing)(ys)
pined_chunk   xs ys = filter(\y -> (locate_pin xs y) /= Nothing )(ys)

locate_pin :: Pin    ->   -- Single pin.
              Sequence -> -- Input sequence.
              Maybe Int   -- position of the in the sequence.
locate_pin  xs ys = locate_pin' xs ys 0
locate_pin' xs [] n = Nothing
locate_pin' xs (y:ys) n
   | isPrefixOf xs (y:ys) = Just n
   | otherwise  = locate_pin' xs ys (n+1)

pin_value :: Maybe Int -> Int
pin_value (Just x) = x
pin_value Nothing = 0

fun_align ::  [Sequence ] ->
              [Sequence ]
fun_align [] = []
fun_align xs = [ add_d x | x <- xs]
   where
 m =  maximum [length x | x <- xs ]
 add_d ::  Sequence  -> Sequence
 add_d x
  | length x == m = x
  | otherwise = x ++
            concat (replicate (m - length x) p)
   where
    p1 =concat (replicate (m - length x) p)
            p = [ 8 ]

lead_function :: Sequence -> [Sequence]  -> [Sequence]
lead_function [] [] = []
lead_function  xs []  = []
lead_function  [] ys  = ys
lead_function   xs  ys =  map (lead_function' xs [] ) ys

lead_function' [] align [] = align
lead_function' xs align [] =align
lead_function' [] align ys =(reverse align) ++ ys
lead_function' (x:xs) align (y:ys)
   | x ==y = res
   | otherwise = res1
   where
   res = lead_function' xs (x:align) ys
   res1 =lead_function' xs (9:align) (y:ys)



test_align_length :: [Sequence]  -> Bool
test_align_length [] = True
test_align_length xs
            | all(\x->(length x) == length (head xs)) xs = True
    | otherwise =False


 ---------------------------------------------------------------
--  FUNCTION TO GENERATE RANDOM  SET OF SEQUENCES           --
---------------------------------------------------------------
mkRandom :: Int ->
```

```
            Int ->
    IO [[Int]]
mkRandom m n = do
         let
          g = mkStdGen 1701
          cs :: [Int]
          cs = randoms g
          cs0 = map  (`mod` 4) cs
          mk_grid' 0 _ _ res = res
          mk_grid' m n l res = mk_grid' (m-1) n l2 (l1:res)
                    where (l1, l2) = splitAt n l

          grid = mk_grid' m n cs0 []
         return grid

convert :: [[Int]] -> [Sequence1]
convert [] = []
convert (x:xs) = map dd x : convert xs
        where
           dd 0 = A
           dd 1 = C
           dd 2 = G
           dd 3 = C
           dd 8 = D -- This character is used for deletion
           dd 9 = I -- The character is used for insertion


main =  do args <- getArgs

           let
     n = read (args!!0)  -- Number of input sequences
     l = read (args!!1)  -- length of each input sequence

     xs <-  mkRandom  n l
   let
            m =  align_chunk  xs
            res = convert m
           print( res )
```

A.2 Complete Parallel Code of Version IIa

This section presents the code of the best version of genetic alignment program which delivers a best performance on both architectures.

```
module Main where
import System(getArgs)
import GlaExts(trace)

import List
import Random
import Strategies
data Aminoacid = A | C | U | G | D |I
  deriving (Read,Show,Eq,Ord)

type Sequence1 = [Aminoacid]

type Sequence =  [Int]
```

```
type Pin =  [Int]
type SubSequence =  [Int]

-- This  accepts the chunk of sequences to be aligned and produces an alignment
-- by calling the two top level functions  " Bestpin" and  "divide"
align_chunk ::  [Sequence] -> -- List of input sequences.
                [Sequence]   -- Aligned sequences.
align_chunk [] = []
align_chunk  xs  = fun_align all_res
     where
      best = Bestpin xs
      all_res = divide xs best

-- This divide takes  a list of sequences and a best pin for a given list split
-- it  uses  a pin  into three chunks left , right ,and unpinned chunk to be
-- aligned independently  by concurrent calling between align_chunk and divide
-- functions
divide ::  [Sequence] -> -- List of input sequences
            Pin -> -- Best pin
   [Sequence]   -- List of aligned sequences.
divide [] [] = []
divide xs [] = xs -- this  represents the basic alignment to the sequence
divide xs pin = (combine pin_var res_lift     res_right     res_unpinch )
           `demanding` strategy
        where

        (rightch,leftch,unpinch1) = splitting_sequences pin xs
        unpinch = lead_function pin unpinch1
        res_unpinch = align_chunk  unpinch
        res_right = align_chunk    rightch
        res_lift = align_chunk    leftch
strategy = rnf res_lift `par`
          rnf res_right  `par`
  rnf res_unpinch
combine :: Pin -> [Sequence] ->[Sequence] ->[Sequence] -> [Sequence]
combine  pin left_seqs right_seqs unpinned_seqs
         = ( zipWith ( cat_sequence pin) left_seqs right_seqs) ++ unpinned_seqs           where
           cat_sequence :: Sequence -> Sequence -> Sequence -> Sequence
   cat_sequence pin ls rs = ls ++ pin ++ rs


-- The Bestpin function takes a list of sequences and produces best as output

Bestpin ::  [Sequence] -> -- List of input sequences.
            Pin -- Best pin.
Bestpin []= []
Bestpin xs =  best_pin   pins_dis
  where
    all_substring  = par_substring_sequences xs xs
    pins_with_occurrence =   all_substring
    pins = map  fst( extract_max_pins pins_with_occurrence )
    extract_longest_pins = longest_pin  pins
    pins_dis =  pin_average_distance   extract_longest_pins   xs


---------------------------------------------------------------
-- ALL FUNCTIONS CALLING BY Bestpin FUNCTION       --
---------------------------------------------------------------
substring_sequences :: [ Sequence] ->  Sequence -> -- The input sequences
                       [(Sequence,Int )]      -- list of substring list
```

```
substring_sequences ys x = nub res1
          where
   res =   subseq x
   res1= form_pins res  ys

par_substring_sequences ::   [Sequence ] ->[Sequence ] ->[(Pin,Int )]
par_substring_sequences   xs ys =
        foldr (++) [] (parMap rnf (substring_sequences ys) xs)
subseq :: Sequence    -> -- A single sequences from the input sequences
          [SubSequence ]    -- all substring from the input sequences
subseq (x:xs) = inits (x:xs)++  subseq xs
subseq [] = []


Form_pins :: [SubSequence] -> -- A list of all substrings of a single sequence.
             [Sequence ]  -> -- A list of input sequences
   [(Pin,Int )]   --List of pins and its occurrence.
Form_pins [][] = []
Form_pins[] ys = []
Form_pins(x:xs) ys
   | num > 1 =  (x,num): form_pins xs ys
   | otherwise = form_pins xs ys
   where
   num = form_pins' x ys

form_pins' :: SubSequence    ->  -- Single element from substring list
              [Sequence]  ->  -- A list of input sequences
       Int            -- Number of occurrences as pins
form_pins' [] [] = 0
form_pins' m []  = 0
form_pins' m (x:xs)
  |((check_for_snd_appears == Nothing ) &&
    (check_for_appears /= Nothing ))
     = 1+ form_pins' m xs
  |otherwise =  form_pins' m xs
  where
   check_for_appears   = locate_pin m x
   where_pin_appears = pin_value(check_for_appears)
   reset_of_sequence  = drop (where_pin_appears + length m ) x
   check_for_snd_appears =  locate_pin  m reset_of_sequence

extract_max_pins  :: [(Pin,Int )] -> [ (Pin ,Int )]
extract_max_pins [] =[]
extract_max_pins ((p,n):xss) =   foldr (extract_max_pins')  [(p,n)]  xss


extract_max_pins' :: (Pin ,Int)  ->[ (Pin,Int)] ->  [(Pin,Int)]
--extract_max_pins'   []   []  = []
-- extract_max_pins'  aa_pin  []   = aa_pin
extract_max_pins'  (p,n) aa_pin@((p',n'):_)
     | n' >  n  =   aa_pin
     | n' ==  n  = aa_pin ++ [(p,n)]
     | otherwise = [(p,n)]

longest_pin  ::  [Pin] ->[Pin]
longest_pin [] = []
longest_pin (xs:xss) = foldr  (longest_pin') [xs] xss

longest_pin' ::  Pin  ->[Pin] ->[Pin]
longest_pin' pin  xs@(pin':_)
```

```
     | length pin' >  length pin   = xs
     | length pin' ==  length pin  = (pin:xs)
     |  otherwise = [pin]


pin_average_distance ::  [Pin]  ->   -- A list of pins
                       [Sequence]  ->  -- A list of input sequences
                       [(Pin, Int ) ]


pin_average_distance  [] [] = []
pin_average_distance [] _ = []
pin_average_distance _ [] = []
pin_average_distance   xss   yss = [ (xs ,((sum (map (
    pin_average_distance'  xs) yss ) `div` length yss )))| xs <- xss]
    where
    pin_average_distance'  :: Pin   -> Sequence ->   Int
    pin_average_distance' [] [] = 0
    pin_average_distance' xs ys
      | check_res == Nothing  = 0
      | otherwise = abs(((length ys) `div` 2) -
       ((pin_value ( check_res )) + (length xs  `div` 2)))
        where
 check_res = locate_pin xs ys

best_pin ::  [(Pin,Int)] -> Sequence
best_pin [] = []
best_pin ((p,n):xs) = fst( best_pin' (p,n)  xs)
best_pin' (p,n) [] = (p,n)
best_pin' (p',n') ((p,n):xs)
  |  n <= n'   =   best_pin' (p,n) xs
  | otherwise = best_pin' (p',n') xs


---------------------------------------------------------------
-- ALL FUNCTIONS CALLING BY divide  FUNCTION                 --
---------------------------------------------------------------

splitting_sequences :: Pin  -> -- Best pin
                    [Sequence] -> -- List of input Sequences
   ([Sequence] , [Sequence] , [Sequence])
splitting_sequences [] [] = ([], [] ,[])
splitting_sequences [] ys = (ys, [], [])
splitting_sequences xs ys  = (right_chunk , lift_chunk , unpinned)
       where
        right_chunk = right_sequence xs ys
        lift_chunk = left_sequence xs ys
        unpinned = unpinned_chunk xs ys


right_sequence :: Pin  -> -- best pin.
                 [Sequence] -> -- List of input sequences.
         [Sequence]   -- List of right sequences.

right_sequence [] [] = []
right_sequence [] ys = ys
right_sequence xs ys = map (right_sequence' xs ) (pined_chunk xs ys)
    where
    right_sequence' :: Pin ->  Sequence -> Sequence
    right_sequence' [] [] = []
```

```
    right_sequence' [] ys = ys
    right_sequence' xs ys
       | (num_pin + length xs)  == length ys  = []
       | check_res /= Nothing = drop (  num_pin + (length  xs ))  ys
       where
       check_res =  locate_pin xs ys
       num_pin  = pin_value( check_res)
left_sequence :: Pin  -> -- best pin.
                [Sequence] -> -- List of input sequences.
        [Sequence]   -- List of left sequences.
left_sequence [] [] = []
left_sequence [] ys = ys
left_sequence xs ys = map (left_sequence' xs ) (pined_chunk xs ys)
    where
     left_sequence' :: Pin ->  Sequence -> Sequence
     -- This function  extracts the left sequence from single sequence
     -- the inputs are best pin + single sequence.
     left_sequence' [] [] = []
     left_sequence' [] ys = ys
     left_sequence' xs ys
        | num_pin  > 0 = take ( num_pin ) ys
        | otherwise = []
where
 num_pin =   pin_value (locate_pin xs ys)

unpinned_chunk    xs ys = filter(\y -> (locate_pin xs y) == Nothing)(ys)
pined_chunk   xs ys = filter(\y -> (locate_pin xs y) /= Nothing )(ys)

locate_pin :: Pin       ->  -- Single pin.
              Sequence  ->  -- Input sequence.
              Maybe Int     -- position of the in the sequence.
locate_pin  xs ys = locate_pin' xs ys 0
locate_pin' xs [] n = Nothing
locate_pin' xs (y:ys) n
   | isPrefixOf xs (y:ys) = Just n
   | otherwise  = locate_pin' xs ys (n+1)

pin_value :: Maybe Int -> Int
pin_value (Just x) = x
pin_value Nothing = 0


fun_align ::  [Sequence ] -> -- List of input sequences to aligned
              [Sequence ]
fun_align [] = []
fun_align xs = [ add_d x | x <- xs]
       where
       m =  maximum [length x | x <- xs ]
       add_d ::  Sequence  -> Sequence
       add_d x
         | length x == m = x
   | otherwise = x ++ concat (replicate (m - length x) p)
   where
     p1 =concat (replicate (m - length x) p)
     p = [ 8 ]

lead_function :: Sequence -> [Sequence]  -> [Sequence]
lead_function [] [] = []
lead_function  xs []  = []
lead_function  [] ys  = ys
```

```
lead_function   xs  ys =  map (lead_function' xs [] ) ys

lead_function' [] align [] = align
lead_function' xs align [] =align
lead_function' [] align ys =(reverse align) ++ ys
lead_function' (x:xs) align (y:ys)
   | x ==y = res
   | otherwise = res1
   where
   res = lead_function' xs (x:align) ys
   res1 =lead_function' xs (9:align) (y:ys)



test_align_length :: [Sequence]  -> Bool
test_align_length [] = True
test_align_length xs
            | all(\x->(length x) == length (head xs)) xs = True
    | otherwise =False


 --------------------------------------------------------------
--  FUNCTION TO GENERATE RANDOM  SET OF SEQUENCES           --
---------------------------------------------------------------

mkRandom1 :: Int -> -- Input value represents the number of sequences
            Int -> -- Input value represents the length of each sequence
    IO [[Int]] -- List of random input sequences

mkRandom1 m n = do
         let
          g = mkStdGen 1701 -- deterministic input via fixed seed val
          cs :: [Int]
          cs = randoms g
          cs0 = map  (`mod` 4) $ cs


          mk_grid' 0 _ _ res = res
          mk_grid' m n l res = mk_grid' (m-1) n l2 (l1:res)
                    where (l1, l2) = splitAt n l

          grid = mk_grid' m n cs0 []
         return grid

convert :: [[Int]] -> [Sequence1]
convert [] = []
convert (x:xs) = map dd x : convert xs
        where
           dd 0 = A
           dd 1 = C
           dd 2 = G
           dd 3 = C
           dd 8 = D -- This character is used for deletion
           dd 9 = I -- The character is used for insertion


main =  do args <- getArgs

           let
     n = read (args!!0)  -- Number of  input sequences
```

```
     l = read (args!!1)  -- length of each input sequence
   xs <-  mkRandom1  n l
   let
             m  =  align_chunk xs
             res = convert m
           print( res )
```

A.3 Finite Map Code

The following shows all modification made to implement the Finite Map library in the genetic alignment program.

```
import FiniteMap(FiniteMap,listToFM ,emptyFM,addListToFM
                   ,addListToFM_C,fmToList,delListFromFM
                    ,delFromFM ,sizeFM,lookupFM,foldFM,
                     plusFM_C ,keysFM)
----------------------------------------------------------
-- ALL FUNCTIONS CALLING BY Bestpin FUNCTION    --
------------------------------------------------------------
substring_sequences :: [ Sequence] ->       -- The input sequences
                       [(SubSequence, Int)] -- List of substring list
substring_sequences [] = []
substring_sequences xs = res_pin
       where
        res_pin' = substring_sequences' xs
        res_pin = form_pins (cat_listoffm res_pin')
substring_sequences' :: [ Sequence]  ->
                        [FiniteMap SubSequence Int]
substring_sequences' []  = []
substring_sequences' (x:xs)  = fm_of_critical : substring_sequences' xs
     where
      all_substrings = subseq x
      fm_of_substring = list_of_substring_fm all_substrings
      el_double_substring = critical_function fm_of_substring
      fm_of_critical = delListFromFM fm_of_substring el_double_substring

-- This is to filter substrings which occur once in FM.
critical_function fm = filter (\ x -> case lookupFM fm x
                              of Just n -> n>1) (keysFM fm)

list_of_substring_fm :: [SubSequence ] -> FiniteMap SubSequence Int
list_of_substring_fm [] = emptyFM
list_of_substring_fm xs = addListToFM_C (+) emptyFM [(x,1) | x<-xs]

cat_listoffm :: [FiniteMap SubSequence Int] -> FiniteMap SubSequence Int
cat_listoffm ys = foldr (plusFM_C (+)) emptyFM ys

form_pins ::  FiniteMap SubSequence Int ->  [(Pin ,Int)]
form_pins  ys = pins_in_list
   where
   list_of_cri_substrings = critical_function1 ys
   pins_in_fm = delListFromFM ys list_of_cri_substrings
   pins_in_list = fmToList  pins_in_fm
-- This is to filter substrings which occur once in FM.
critical_function1 fm = filter (\ x -> case lookupFM fm x
                                of Just n -> n==1)   (keysFM fm)
```

Bibliography

[1] D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):pages 123–169, 1998.

[2] A. Chien, J. Dolby, B Ganguly, V Karamecheti, and X. Zhang. High Level Parallel Programming: the Illinois Concert System. Technical Report Illinois 61801, Computer Science, University of Illinois, 1998.

[3] L. Chamberlain and E. Christopher. ZPL A Machine Independent Programming Language for Parallel Computers. *IEEE Transaction on Software Engineering*, 26:pages 197–212, March 2000.

[4] H. Loidl. Load Balancing in a Parallal Reducer. In *Trends in Functional Programming*, volume 3, pages 63–75. Intellect Ltd, 2002. ISBN 1-84150-070-4.

[5] P. Trinder, J. Barry, M. Davis, K. Hammond, S. Junaidu, U. Klusik, H. Loidl, , and S. Peyton Jones. GpH: An Architecture-Independent FunctionalLanguage. Unpublished, http://www.cee.hw.ac.uk/ dsg/gph/papers/abstracts/arch-indep.html, July 1998.

[6] K. Hammond, H. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *Conference on High Performance Functional Computing*, pages 208–221, Denver, Colorado, April 1995.

[7] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[8] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace*, 1997.

[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, USA, 1994.

[10] J. Jack, R. Hempel, H. Anthony, and D. Walker. A Proposal for a User-Level Message-Passing Interface in a Distributed Memory Environment. Technical Report TM-12231, University of Tennessee, Knoxville, TN, USA, 1992.

[11] H. Dietz. Linux Parallel Processing HOWTO, January 1999. Midwest Workshop on Parallel Processing, Kent State University, http://yara.ecn.purdue.edu/ pplinux/pphowto.

[12] D.E Culler and J. P. Singh. *Parallel Computer Architecture: a Hardware/Software Approach* . Morgan Kaufmann, 1999.

[13] M. Jones and P. Hudak. Implicit and Explicit Parallel Programming in Haskell. Technical Report CT 06520-2158, Department of Computer Science, Yale University, August 1993.

[14] P. Roe. *Parallel Programming Using Functional Language* . PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[15] K. Hwang and Z Xu. *Scalable Parallel Computing - Technology, Architecture, Programming* . WCB McGraw-Hill USA, 1998.

[16] R.S. Nikhil. ID Reference Manual. Technical Report CSG Memo 284-2, Laboratory for Computer Science, M.I.T., July 1991.

[17] P. Trinder, K. Hammond, J. Mattson, A. Partridge, and S. Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–88, 1996.

[18] M. Hamdan. *A Combinational Framwork for Parallel Programming Using Algorithmic Skeletons.* PhD thesis, Department of Computing Science, Heriot Watt University, January 2000.

[19] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, MA, 1989.

[20] P. Kelly and F. Taylor. Coordination Languages . In *Research Directions in Parallel Functional Programming*, pages 305–321. Springer, 1999.

[21] C. Koelbel, D. Loveman, and JR. Schreaiber, R.and Steele. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, March 1994.

[22] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler Support for Machine-Independent Parallel Programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[23] N. Gaarder and M. Bruggencate. Openmp:an autotasking perspective. Technical Report MN 55121-1560, USA., Programming Group Silicon Graphics, Inc., January 2003.

[24] S. Gregory. *Parallel Logic Programming in PARLOG* . Addison-Wesley, 1988.

[25] D. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, August 1998.

[26] T. Braunl. Parallaxis-III: Architecture Independent Data Parallel Processing. *IEEE Transaction on Software Engineering*, 26:pages 227–244, March 2000.

[27] P. Trinder, H. Loidl, and K. Hammond. Large Scale Funtional Applications. In *Research Directions in Parallel Functional Programming* , pages 399–463, 1999.

[28] D. Suciu and V. Tannen. CoPa: a Parallel Programming Language for Collections, November 2002. University of Pennsylvania, Unpublished, http://citeseer.nj.nec.com/389641.html.

[29] D. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):pages 249–274, Fall 1997.

[30] E. Pontelli. Adventures in Parallel Logic Programming, October 2002. New Mexico State University, Unpublished, http://www.cs.nmsu.edu/ epontell/adventure/paper.html.

[31] S. Peyton Jones. *The Implementation of Functional Programming Language* . Prentice/Hall International , 1986.

[32] R. Plasmeijer and M. Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addisn-Wesley, 1993.

[33] B. Lisper. A Brief Survey of Functional Programming Languages. August 2002. Mlardalen University, Unpublished, http://www.idt.mdh.se/kurser/cd5100/ht02/history.html.

[34] A. Church. A Set of Postulates for the Foundation of Logic. . *Annals of MATH.* , 33:pages 346–366, 1932.

[35] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation. *Part I COMM, ACM* , 3:pages 184–195, 1960.

[36] J. Backus. Can Program be Liberated from the Von Neumann Style? A Functional Style and Its Algerbra of Programs . *Communications of ACM* , 21:pages 613–641, 1978.

[37] D. Turner. Miranda: A non-strict Functional Language with Polymorphic Types. , September. In Proceedings of Functional Programming Languages and Computer Architecture, J.P.Jouannaud (Ed),Springer-Verlag, Vol 201. 31, 1985 .

[38] L. Augustsson. A Compiler for Lazy ML. . *Proceedings of the ACM Symposium on Lisp and Functional Programming, Austin, Texas, USA.* , pages 218– 227, 1984.

[39] P. Hudak, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. A Report on the Functional Programming language Haskell, Version 1.2. *ACM SIGPLAN Notices 27(5)*, 1992.

[40] S. Junaidu. *Parallel Functional Language Compiler for Message Passing Multicomputers*. PhD thesis, Department of Computing Science, University of St Andrews , March 1998.

[41] P. Trinder, K Hammond, H. Loidl, and S. Peyton. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):pages 23–60, January 1998.

[42] G. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, School of Computer Science Carnegic Mellon University, April 1993.

[43] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña Marí. The Eden Coordination Model for Distributed Memory Systems. In *HIPS'97 — High-Level Parallel Programming Models and Supportive Environments*. IEEE Press, 1997.

[44] S. Thompsom. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.

[45] J. Peterson and O. Chitil. Glasgow Parallel Haskell, A Purely Functional Language . December 2002. Unpublished, http://www.cee.hw.ac.uk/~dsg/gph/.

[46] H. Loidl, P. Trinder, K. Hammond, S. Junaidu, R. Morgan, and S. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11(12):pages 701–752, October 1999.

[47] M.P. Jones. Hugs 1.3 The Haskell User's Gopher System User Manual. Technical Report NOTT-CS-TR-96-2, Nottingham University , August 1996.

[48] P. Trinder, H. Loidl, and K. Hammond. The Multi-Architecture Performance of the Parallel Functional Language GPH. In Bode, A. and Ludwig, T. and Wismüller, R., editor, *Euro-Par 2000 — Parallel Processing*, volume 1900 of *LNCS*, pages 739–743, Munich, Germany, 29.8.-1.9., 2000. Springer-Verlag.

[49] S. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *European Symposium on Programming*, pages 18–44, 1996.

[50] P. Sansom and S. Peyton Jones. Time and space profiling for non-strict higher-order functional languages. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–366, San Francisco, California, 1995.

[51] H. Loidl, P. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4):471–486, 2001. Selected papers from HLPP'01 — International Workshop on High-level Parallel Programming and Applications, Orleans, France, 26-27 March, 2001.

[52] P. Pepper and M. Südholt. Deriving Parallel Numerical Algorithms using Data Distribution Algebras: Wang's Algorithm. In *HICSS'97 — 30th Hawaii International Conference on System Sciences*, pages 7–10, Hawaii, USA, January 7–10, 1997. IEEE.

[53] J. O'Donnell and G. Rünger. Abstract Parallel Machines. *Computers and Artificial Intelligence*, 19:105–129, 2000.

[54] A. Abdallah. Functional Process Modeling . In *Research Directions in Parallel Functional Programming*, pages 339–360. Springer, 1999.

[55] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, 1989.

[56] J. Blazewicz. *Handbook on Parallel and Distributed Processing*. International Handbooks on Information Systems. Springer, 2000.

[57] D. Sankoff. *Time Warps, Spring Edits and Macromolecules: the theory and practice of sequence comparison* . Addison-Wesley, 1983.

[58] H. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

[59] G. Amdahl. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Press*, 30:pages 483–485, 1967.