Parallelization of the AAE algorithm

By

Akram Hameed (BA-BComp.)

A dissertation submitted to the
School of Computing
In partial fulfilment of the requirements for the degree of

Bachelor of Computing with Honours

UNIVERSITY OF TASMANIA

November, 2007

DECLARATION

I hereby declare that to the best of my knowledge, this thesis has not been submitted for the award of any diploma or degree at any other tertiary institution. It is also my belief that the thesis contains no previously published material except where due reference is made.

…………………………

Akram Hameed

Abstract

The exact algorithm formulated by Kececioglu and Starrett (2004) provides a solution to the NP-complete problem of aligning alignments in most biological cases. This work investigates potential speedups that may be gained through the parallelization of the dynamic programming phase of this particular algorithm. Results indicate it is possible to improve the run time performance of the algorithm over non-trivial alignments that compose to a matrix of greater than 10^5 cells.

ACKNOWLEDGEMENTS

Professor Sale gave me the opportunity to perform this research; so, first thanks go to him for being so optimistic about my ability to do this. Second, huge thanks to Phil for allowing me to bounce questions off him. Went out of his way to explain things I found perplexing and was always there when I needed to ask a question.

To the academic staff and especially Jacky: your enthusiasm made something that might have felt like 'just another year' into an eye-opening experience. Please keep up the good work.

And you, gentlemen all of my office – you whom tolerated my vitriolic conversations with my taciturn console yet spoke not a word of annoyance yourselves…the office was indeed full of *win*. Good luck to us all.

My parents were kind enough to do some proofing on this document, as well as lending me a car when I needed to work all night; I owe you guys more than I can repay, I think!

Thanks also go to whoever ended up being in charge of Gundam while the rest of us were busy writing thesis (obscure?).

Finally, I thank my loving wife for having the temperance to hang around while I finished writing; this may not have taken as long to write as a pHD thesis but I'm betting it felt like one to her at times. Thanks M. ☺

TABLE OF CONTENTS:

Declaration i

Abstract ii

Acknowledgements iii

Table of contents: iv

Table of Figures vii

List of Tables viii

1. Introduction & objective 1

2. Literature Review 2

2.1 Overview: 2

2.2 Sequence comparison background issues: 2

2.2.1 Background biological concepts: 2

2.2.1.1 DNA and Molecular Sequences: 2

2.2.2 Parallel and Distributed systems: 4

2.2.2.1 Types of Parallel System: 4

2.2.2.2 On the Parallelization of Algorithms: 5

2.2.3 Objectives of Multiple Sequence Alignment: 6

2.2.4 Origins of Sequence alignment: 7

2.3 Sequence Alignment Theory: 8

2.3.1 Pair-wise Sequence Alignment: 8

2.3.1.1 Elements of alignment algorithms: 9

2.3.1.2 Substitution Matrices and the Objective Function: 9

2.3.1.3 Dynamic programming with Needleman-Wunsch: 11

2.3.2 Multiple Sequence Alignment: 12

2.3.2.1 Dynamic Programming: 12

2.3.2.2 Progressive Alignment: 13

2.3.2.3 Iterative Alignment: 13

2.3.2.4 Consistency-based Alignment: 14

2.4 The Aligning Alignments Exactly Algorithm: 15

2.4.1 The problem: 15

2.4.2 The exact algorithm (Summary): 17

2.4.3 Benchmark Test Suites: 20

2.5 Existing applications of parallel sequence alignment: 20
2.6 Summary: 22
3. Approaches to Parallelization 22
3.1 Considered elements 22
3.1.1 Dynamic programming table generation 22
3.1.2 Dominance and Bounds pruning 23
3.1.2.1 Space reduction under Hirschberg's principle 24
3.1.2.2 Dominance Pruning 24
3.1.2.3 Bound Pruning (an overview) 26
3.2 Logic & Pragmatism 27
3.2.1 Subdividing the Dynamic Programming Matrix 27
3.2.1.1 Blocking as an efficiency enhancer 27
3.2.1.2 Striping to maximise productivity 27
3.2.1.3 Using stripes and blocks to improve timeliness 28
3.2.1.4 Speedup of the exact algorithm. 31
3.2.1.5 Parallel exact algorithm walkthrough 34
4. Method 35
4.1 Introduction 35
4.2 Metrics 36
4.3 Experiment Design 37
4.3.1 System architecture 37
4.3.2 Assumptions 39
4.3.2.1 Networking 40
4.3.2.2 Object Serialization 41
4.3.3 Calculation of results 41
4.3.3.1 Preprocessing of data with ClustalW 41
4.3.3.2 Serial AAE 42
4.3.3.3 Parallel AAE 43
4.3.4 Summary 44
5. Results and Observations 44
5.1 Performance comparison 44
5.1.1 Parallelization overhead 44
5.1.2 Small alignments 46
5.1.3 Medium alignments 49

5.1.4 Large alignments 52
5.1.5 Speed up in general 55
5.2 Discussion 56
5.2.1 Parallel algorithm performance 57
5.2.2 Thresholds and Unpredictability 59
5.2.2.1 Unexpected outcomes 60
5.2.2.2 Variable block sizes 61
5.2.3 Summary 61
6. Conclusions 62
7. Further work 63
8. References: 64
9. Appendices 67

TABLE OF FIGURES

Figure 1:DNA double helix showing base pairing and generic structure. 3
Figure 2: Shape propagation in the dynamic programming matrix................................ 23
Figure 3: A example of a 5x5 block with numbered cells... 29
Figure 4: A 4x4 block with logical cached column highlighted 30
Figure 5: Input row from a 5x5 block 31
Figure 6: 6 stripe by 19 block overlay matrix. Darker shades indicate longer waits for jobs 33
Figure 7: Communication relationship between master and slave cluster 38
Figure 8: 13th instance of RV11, performance over pUnit scaleup................................ 47
Figure 9: 28th instance of RV11 performance over pUnit scaleup................................. 48
Figure 10: 4th instance of RV12, performance over pUnit scaleup................................. 48
Figure 11: 15th instance of RV11, performance over pUnit scaleup............................... 50
Figure 12: 43rd instance of RV12, performance over pUnit scaleup............................... 51
Figure 13:18th instance of RV12, performance over pUnit scaleup................................ 51
Figure 14: 38th instance of RV12, performance over pUnit scale up............................. 52
Figure 15: 7th instance of RV12, performance over pUnit scale up............................... 53
Figure 16: 7th instance of RV12, performance over pUnit scale up............................... 54

LIST OF TABLES

Table 1: Speedup factor as a function of mean processing time over RV11 45
Table 2: Speedup factor as a function of mean processing time over RV12 45
Table 3: Mean run time and speed up over RV1 dataset (‘Small’ range) 55
Table 4: Mean run time and speed up over RV1 dataset (‘Medium’ range) 55
Table 5: Mean run time and speed up over RV1 dataset (‘Large’ range) 56

1. INTRODUCTION & OBJECTIVE

Molecular sequence alignment is a field of bioinformatics that is generally concerned with determining relationships between various organisms (Durbin et al. 1998). This is achieved by analysing the similarities between protein and nucleotide sequences and deriving conclusions about their phylogenetic or even functional properties. This field is not new; it has existed for over thirty years – however recently it became apparent that a sub-problem within the field could be rendered soluble given current technology. The problem was that of 'aligning alignments' – the composing of an alignment of many sequences of amino acids or nucleotides from two pre-existing alignments(Kececioglu, J & Zhang 1998). Despite being proven to be NP-complete (Wang & Jiang 1994), Kececioglu and Starrett formulated an algorithm that solves aligning alignments in polynomial time (2004). Over non-trivial problems, however, the solution may have room for improvement, which leads to the objective of this work:

Investigate the potential for the parallelization of the Aligning Alignments Exactly algorithm and determine the extent to which any parallelization is worthwhile.

Given this objective, the following thesis documents research performed upon the exact algorithm with the intention of improving its timeliness through parallelization. Initially, consideration is given to previous work in the field. Following this, an investigation on the pragmatism of parallelization is presented. As a result of the work undertaken for that section, it was determined that the algorithm could most practicably be parallelized by exploiting the generation of a dynamic programming matrix that forms the core time-consuming activity of the algorithm[1]. The approach adopted for examining the extent to which a speedup could be achieved involved the application of the exact algorithm's logic in such a manner as to parallelize its core functions so they may be executed as a distributed application. Tests[2] were performed over a number of machines in a micro-cluster in order to determine the extent to which the hypothesis could be satisfied. The results section[3] details the outcome of this research, with a discussion on elements of the findings and a conclusion[4] on their validity. Finally, areas that would benefit from further work are mentioned[5].

[1] See section 3.1.1
[2] See section 4.3
[3] See section 5
[4] See section 6
[5] See section 7

2. LITERATURE REVIEW

The first of several chapters investigating the objective stated in the introduction, this literature review does not presume knowledge of the field of sequence alignment, nor that of parallelization. It does, however, make assumptions about the minimum level of understanding that the reader will have; namely in their comprehension of algorithmic complexity issues.

2.1 OVERVIEW:

The intention of this chapter is to provide grounding in the literature surrounding multiple sequence alignment. In particular, it examines the Aligning Alignments Exactly algorithm (hereafter referred to as 'AAE') conceived of by Kececioglu and Starrett for the exact alignment of multiple alignment pairs. Parallelization issues are also discussed in order to support the investigation of the proposed hypothesis that it may be probable to extract a performance enhancement through the parallelization of the AAE algorithm.

2.2 SEQUENCE COMPARISON BACKGROUND ISSUES:

This section considers some of the background concepts of sequence comparison and their origins. A brief overview of the history of sequence alignment is presented with the intention of providing a context for current techniques. Additionally, an introduction to DNA and protein sequences establishes the problem domain for the research issue.

2.2.1 BACKGROUND BIOLOGICAL CONCEPTS:

2.2.1.1 DNA and Molecular Sequences:

Deoxyribonucleic acid (DNA) molecules play a vital role in the functioning of organic life-forms. Contained within the cells of all known organisms, DNA encodes the information necessary for the production of proteins that enable cells to function and replicate and regulates the articulation of said proteins to produce an organism's genetic individuality (Feitelson & Treinin 2002). As this review concerns issues related to sequence comparison algorithms, the reader may find further explanation of the exact processes of cellular function and DNA in

(Alberts et al. 2002). The alignment of molecular sequences is traditionally performed upon nucleotide[6] or protein sequences.

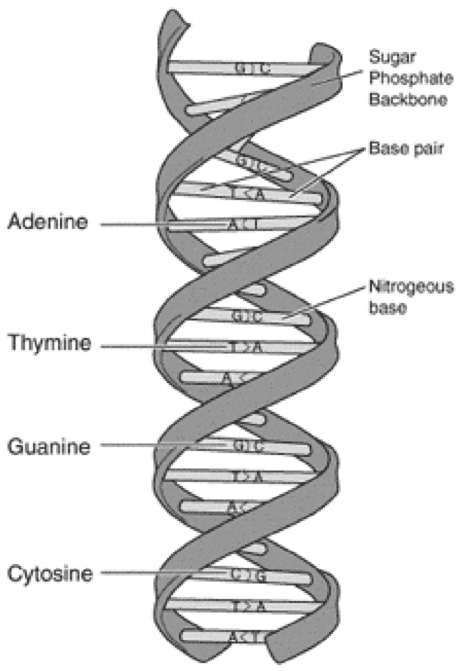

Figure 1:DNA double helix showing base pairing and generic structure.

Fig 1. shows a simplified representation of the model of DNA that Watson and Crick proposed (1953). Visible are the nitrogenous bases that comprise the articulated 'alphabet' of DNA. It is these bases, Adenine (A), Thymine (T), Guanine (G), and Cytosine (C) which pair to form the structure of DNA.

As mentioned, they form an alphabet of sorts. If the English alphabet contains 26 characters, the DNA base alphabet contains just the four specified. Both nucleotide and protein sequences are represented by long text strings. Evolutionary events such as the insertion, substitution or deletion of genetic material are modelled by placing gaps in the place of characters that prevent a 'best' alignment (Mount 2001). Before continuing, it would be instructive to mention that protein sequences are sequences of amino acids which themselves are coded from DNA bases; one example might be the amino acid 'Arginine' which can be coded from cytosine, guanine and adenine or more simply CGA. Such a triplet of nucleotides in a particular coding sequence is called a codon (King & Stansfield 1997). There exist 20 amino acids that are biosynthesized in protein sequences, each of which have an English character assigned to them for the purposes of protein sequence analysis and alignment (Campbell, Reece & Meyers 2006).

[6] Nucleotide, *n*. One of the monomeric units from which DNA or RNA polymers are constructed, consisting of a purine or pyramidine base, a pentose, and a phosphoric acid group (King & Stansfield 1997).

Therefore, the comparison and subsequent alignment of biological sequences with respect to this research refers to the comparison of nucleotide or protein sequences. Important issues that should be articulated before proceeding are:

- Given a series of sequences to align, it may become necessary to substitute characters to obtain the 'best' alignment. The caveat of this particular operation is covered in section 2.3.1.1.
- Nucleotide sequences are often measured in terms of base-pairs (bp), the number of which may dictate the mechanism used for sequence alignment.
- The optimality of an alignment of sequences is dependent on several factors that are discussed on section 2.3.

2.2.2 PARALLEL AND DISTRIBUTED SYSTEMS:

The parallelization of a task refers to the subdivision of a given problem into *n*-units which may be processed in parallel to produce a satisfactory solution (Wilkinson & Allen 1998). The notion of dividing work up to be computed in parallel is not a new one. A classic example is the work of Holland, who conceived of a system which might allow for the simultaneous execution of many sub-programs, therefore foreshadowing contemporary operating systems with his vision (1959). In the context of sequence alignment, the hypothesis stipulates that it may be possible to obtain performance gains by parallelizing the AAE algorithm. Prospective gains, however, are dependent on the resolution of several issues which the following sections introduce.

2.2.2.1 Types of Parallel System:

Algorithmic parallelization generally falls between two poles of data parallelization and task parallelization. The former, data parallelization refers to a given data source being processed using the same technique on each node in the parallel architecture (whether those nodes are whole systems or just processors) (Wilkinson & Allen 1998, p. 5). An example of data parallelization is the Seti@home project: each computer running the Seti screensaver performs the same algorithm on its discrete portion of data as do the other 'nodes' in the distributed network (Anderson et al. 2002). Task parallelization refers to the alternative scenario where several nodes may execute different algorithms on the same data source, or alternatively on several data sources. Such parallelization is present in the functioning of multiple-core PCs presently, provided that the operating system supports the execution of two tasks in parallel. The historical alternative to task parallelization has been complicated time-sharing arrangements, whereby a system gives the impression it is running several programs concurrently, when in truth – the processor may only execute a single instruction at a time (Zomaya 1996, p. 6).

2.2.2.2 On the Parallelization of Algorithms:

The primary concern of the hypothesis is the possibility of formulating a parallel version of the AAE algorithm that displays a worthwhile improvement in processing time. Therefore, consider the factors involved in parallelizing algorithm. The speedup factor of an algorithm is defined as:

$$S = \frac{\alpha}{\beta}$$

Where $\boldsymbol{\alpha}$ is defined as the run time of the best sequential algorithm and β is the run time of the parallelized algorithm. Zomaya (1996, p. 14) quite astutely states that the best possible improvement would be that of N given N processing units. This presupposes that the problem may be decomposed into N tasks, therefore suggesting that a factor of N is the optimal speedup. Amdahl (1967) theorized that this maximum would be limited by the amount of inherent parallelism in the algorithm itself.

An additional factor that must be considered is that an improvement in computational throughput is often outweighed by the cost of communication between processing units. This is formalized in the ratio:

$$cp_i = \frac{E_i}{C_i}$$

Where cp_i is the communication penalty on processor i, E_i is the time processor i takes to process the algorithm, and C_i represents the time corresponding only to communications between processing units.

Wilkenson and Allen (1998, p. 26) point out that cp_i will have a value dependant on the *granularity* of the parallelism inherent in the algorithm. In this circumstance, granularity may be defined loosely as the amount of work processed before another communication interval is required. Therefore, *coarse* granularity refers to relatively large processing time periods, while *fine* granularity is the opposite; taking only a modest period to process (Wilkinson & Allen 1998). A final issue that is worth considering is resource utilization; especially given that the most effective versions of the AAE algorithm require quadratic space (see section 3.1.2). Depending on how a resource is defined (whether a resource be a processor or memory); the following metric may be of some utility:

$$U = \frac{O(N)}{NT(N)}$$

Where $\boldsymbol{U}$ is the resource utilization factor, $\boldsymbol{O(N)}$ is the number of operations performed by an $\boldsymbol{N}$-processor machine and $\boldsymbol{NT(N)}$ represents the maximum number of operations that might be performed given $\boldsymbol{N}$ processors and $\boldsymbol{T(N)}$ time units (Zomaya 1996, p. 15). Stated simply, $\boldsymbol{U}$ represents a percentage of resources utilized during a given parallel process. Clearly, when designing a parallel algorithm, time must be spent considering these issues in order to maximize expectant outcomes.

2.2.3 OBJECTIVES OF MULTIPLE SEQUENCE ALIGNMENT:

Before considering any algorithmic details, it is instructive to examine the uses of multiple sequence alignment (hereafter referred to as MSA) in the laboratory. Mount (2001, p. 142) suggests that sequence alignment may be used to infer phylogenetic[7] relationships between the sequences being aligned. He also proposes that structural or functional attributes can be determined through rigorous analysis of conserved or similar regions of sequences. Further support is provided by Schmollinger et al (2004, p. 1) who state that sequence alignment has been used recently to determine functional elements of biological sequences. Wholesale explanation of this subject area is beyond the scope of this paper, thus curious readers would be advised to consider the subject matter in the attached

[7] Phylogeny: *n* 'the pattern of historical relationships between species or other groups resulting from divergence during evolution.'(OED 1989) Therefore, a phylogenetic tree is a representation of the evolutionary relationship between several species (Mount 2001).

reference list. The following section examines the origins of sequence alignment and its most influential architects.

2.2.4 Origins of Sequence alignment:

The 1970s saw what Baxevanis and Ouellette describe as an 'explosion' in the number of DNA sequences that became available for research (2001, p. 145). Comparison and alignment of available sequences was theorized as a productive method of analysis. One such algorithm, the *Needleman-Wunsch* algorithm was proposed by Saul Needleman and Christian Wunsch in a seminal 1970 paper. Needleman-Wunsch envisages a dynamic programming approach to sequence alignment, effectively producing what is known as a *global* sequence alignment. In such an alignment, the entirety of a biological sequence (whether protein sequences as in the original paper, or nucleotide sequences such as DNA more recently) is aligned with the intention that further analysis of the combinate sequence will reveal useful genetic relationships. Given that the dynamic programming approach is a key element in the AAE algorithm, section 2.3.1.3 provides a brief overview of the intrinsic nature of the Needleman-Wunsch algorithm.

In 1981, building on Needleman and Wunsch's work and incorporating some tenets of Hirschberg's (Hirschberg 1975) maximal common subsequence theory, Temple Smith and Michael Waterman proposed a method for finding *local* sequence alignments. Such a method is often employed when the evolutionary distance between sequences is large and the probability of highly conserved[8] regions in the sequences is remote. Buoyed by the success of pair-wise alignment researchers conceived of the benefits of comparing several protein or nucleotide sequences, and hence the theory of aligning multiple sequences emerged in the late 1980s. Since then, many researchers (Collins & Coulson 1984; Myers & Miller 1988; Needleman & Wunsch 1970; Smith & Waterman 1981) have attempted to render the problem into an optimally solvable state, though it is arguable that at present, optimal alignments may be achieved only at the expense of great amounts of time and resources (Kececioglu, J & Starrett 2004; Schmollinger et al. 2004). As a computationally feasible alternative, sub-

[8] In this circumstance, the level of conservation of a region belonging to a sequence is the probability that it is composed of elements that have remained constant during evolution. (Baxevanis & Ouellette 2001)

optimal but efficient methods have grown increasingly popular, as will become evident in the subsequent section on MSA strategies.

2.3 Sequence Alignment Theory:

The alignment of nucleotide and protein sequences has been claimed to allow researchers to identify homologous regions in the genetic makeup of many species (Mount 2001). The term 'homology' refers to conclusions that the genetic material examined shares a common evolutionary history, something that is often considered to be a desirable attribute of sequence alignments (Baxevanis & Ouellette 2001). The field may be divided into approximately two sub-fields: first is pair-wise sequence alignment where two sequences are aligned using various mechanisms; second is multiple sequence alignment, where 3 or more sequences are aligned. Alignment algorithms may be divided into several types: those that attempt to construct a mathematically optimal alignment, and those that attempt to construct a best-possible alignment using alignment heuristics. Baxevanis and Ouellette *inter alios* caution that the optimal mathematical alignment is not necessarily the best possible *biological* alignment in terms of simulating expected evolutionary events (2001, p. 151; Mount 2001, pp. 64-65).

2.3.1 Pair-wise Sequence Alignment:

Computational sequence alignment began with pair-wise sequence alignment. The Needleman-Wunsch algorithm (Needleman & Wunsch 1970) allows for *optimal* global alignments to be achieved and in its original form, required approximately $O(nm^2)$ steps in the worst case, and space requirement of $O(nm)$ where n and m represent two molecular sequences, the latter being the shorter of the two (Mount 2001, p. 75; Needleman & Wunsch 1970; Pevzner 2000, p. 98). Despite the fact that the running time and space requirements of this algorithm have been improved by many (Gotoh, Osamu 1982; Myers & Miller 1988; Schwartz et al. 1991) since it was first published, its run-time complexity for something relatively simple like pair-wise alignment suggests that the act of aligning *multiple* sequences would be a significantly more difficult task. This statement is confirmed by multiple analyses from diverse researchers (Carrillo & Lipman 2006; Kececioglu, J & Starrett 2004; Wang & Jiang 1994).

2.3.1.1 Elements of alignment algorithms:

The alignment of molecular sequences tends to require different elements depending on whether one wishes to align protein sequences or nucleotide sequences. Given an alphabet of characters from which sequences may be constructed, alignment algorithms generally define a character to fill the 'gaps' in the sequence which define homologous regions (Durbin et al. 1998, p. 13). Another element that is common is a *substitution matrix*; the substitution of one amino acid (or indeed, the substitution of one DNA base) for another may improve or degrade the optimality of an alignment, but may alter the inherent meaning or function of the genetic code (Gusfield 1997). As such, the substitution matrix is a construct that provides a 'cost' for the substitution of characters. Substitution matrices are usually crafted by experts in molecular biology, rather than computer scientists (Mount 2001). Substitution 'cost' is often used along with a cost defined for inserting a 'gap' to determine the optimality of an alignment. Gaps, however, may run for more than a single character, so algorithms like the Needleman-Wunsch algorithm (1970) also define a *gap-extension cost* which is summed with the *gap-initiation cost* already mentioned, and finally used to compute alignment optimality. Further explanation of these terms and elucidation of the mechanisms used to define them may be found in (Baxevanis & Ouellette 2001; Durbin et al. 1998; Gusfield 1997; Mount 2001).

2.3.1.2 Substitution Matrices and the Objective Function:

As mentioned, substitution matrices are used to calculate the 'cost' of substituting one character for another in sequence alignment. This theory can be traced back to the work of Dayhoff, Eck and Park (1972), who conceived that to model evolutionary events, such as the insertion or deletion of genetic material, it would be of some utility to assign a probability to the likelihood that a pair of sequences were related or not. The point-accepted-mutation (PAM) model of evolution that Dayhoff and his counterparts published is a widely used set of substitution matrices for amino-acid (therefore, protein sequence) comparison. Baxevanis and Ouellette put the definition of PAM quite simply: 'one PAM is a unit of evolutionary divergence in which 1% of the amino acids have been changed.' (2001). The Dayhoff process model, itself a Markov process, dictates that given an amino acid ρ the probability of mutation to any other amino acid in an alphabet Z is independent of any other mutations which may have occurred at other sites in the protein (Mount 2001, pp. 83-84). This assumption that each amino-acid position is equally mutable is open to conjecture and has been challenged (George, Barker & Hunt 1990). Complementary to the concept of substitution matrices, is the objective function that calculates the quality of the obtained alignment. For multiple sequence alignment, a common (and in our case, used in the AAE algorithm) choice has been the *sum-of-pairs* objective function. This is defined by Pevzner as the following (2000, pp. 125-126):

'For a multiple alignment $A = (a_{ih})$, the induced score of pairwise alignment A_{ij} for sequences a_i and a_j is

$$s(A_{ij}) = \sum_{h=1}^{m} d(a_{ih}, a_{jh}),$$

Where d is the *distance* between elements of any alphabet...' Z' given $\mathrm{Z} \cup \{-\}$ (assuming $'-'$ represents a spacer character not in the original alphabet Z.). Given this expression of a pair wise alignment, the *sum-of-pairs score* (SP-score) may be calculated using the following expression:

$$\sum\nolimits_{i,j} s(A_{ij}).$$

Or more simply, according to Gusfield (1997, pp. 343-348), SP-score is equivalent to the sum of the scores of pair wise global alignments induced by some multiple alignment A. The complexity of calculating the SP-score for a worthwhile problem exactly is defined by Wang and Jiang (1994)to be NP-complete. An explanation of how this is expressed is not relevant to the research issue; therefore the enterprising reader may find a full dissertation in the referenced paper. The combination of substitution matrices and objective functions allow the calculation of optimal alignments for multiple sequence alignments, though it is imperative to consider the substitution matrix employed. The utility of a given substitution matrix depends on the hypothetical relatedness of the sequences to be compared (Mount 2001, p. 83).

2.3.1.3 Dynamic programming with Needleman-Wunsch:

In the interests of providing a clearer explanation of the AAE algorithm, an overview of the original Needleman-Wunsch algorithm follows:

Initially, given two sequences A and B of lengths n and m respectively, construct a $n \times m$ matrix that may be denoted as F. For clarity's sake, assert that F consists of n columns and m rows. Let $D(i,j)$ be a function to represent the edit distance between the first i characters of A and the first j. The definition of edit-distance with relation to sequence alignment may be seen as the number of operations required in order to 'merge' the two sequences A and B. Gusfield defines the dynamic programming approach as having three components – 'the *recurrence relation*, the *tabular computation*, and the *traceback*' (1997, p. 217). Furthermore, define the scores for the edit-distance metric to be 1 for mismatches or gaps and 0 for matches. A recurrence relation for $D(i,j)$ may be established as the following:

$$D = \min[D(i-1,j)+1, D(i,j-1)+1, D(i-1,j-1)+t(i,j)],$$

Where $t(i,j)$ is defined as having a value of 1 if $A(i) \neq B(j)$, or a value of zero if $A(i) = B(j)$. Using this recurrence relation, the second component of the dynamic programming approach is satisfied: tabular computation. The simplest method of composing the dynamic programming table is to use the recurrence relation and establish $A(i=0) \& B(j=0) = 0$ (assuming the comparison of elements begins at $i = 1$ and $j = 1$ respectively) and recurse from this point – hence a top-down computation. Gusfield (1997, p. 219) warns that due to the recursive nature of solution, a top-down approach is needlessly inefficient and advocates the *bottom-up* method described by Gotoh (1982). Finally, in order to compute the *optimal* alignment as defined by this problem, a traceback is conducted through the table of data beginning at

$D(n,m)$ and searching back through the matrix, ascertaining at each step precisely which value the current value was calculated from. This is akin to storing a series of pointers from cell to cell – essentially, given the recurrence, when (i,j) is computed, set a pointer from that cell to cell $(i,j-1)$ if $D(i,j) = D(i,j-1) + 1$; set a pointer to $(i-1,j)$ if $D(i,j) = D(i-1,j) + 1$; and set a pointer to $(i-1,j-1)$ if $D(i,j) = D(i-1,j-1) + t(i,j)$ (Gusfield 1997, p. 221). In essence, conduct a traceback storing pointers indicating the direction that the minimal values were found. As mentioned, the dynamic programming approach is utilised in the AAE algorithm – namely the concept of using a table to store sequence identities[9] and conducting a traceback to obtain an optimal alignment.

2.3.2 *MULTIPLE SEQUENCE ALIGNMENT:*

As mentioned, the alignment of multiple sequences is a significantly more difficult problem than that of pair-wise alignment. As such, there are many mechanisms employed to achieve an alignment, each differing in their implementation. The key methods are listed below, though several key elements of MSA are discussed before continuing.

2.3.2.1 Dynamic Programming:

Pair-wise alignment can be achieved through dynamic programming practices with an approximate complexity of between $O(nm)$ and $O(nm^2)$ (Needleman & Wunsch 1970). Multiple sequence alignment is considered to be a significantly more difficult process; if we take our example of using dynamic programming and conceive of a *k*-dimensional version of the original Needleman-Wunsch algorithm, it has been shown to have a space cost of $O(l^k)$ and a time cost of $O(2^k l^k)$ where there are *k* strings (sequences) of length *l* (Durbin et al. 1998, p. 142; Gusfield 1997, p. 344). While cunning attempts have been made by researchers to improve the performance of dynamic programming methods on multiple sequences (Carrillo & Lipman 2006; Gupta, Kececioglu & Schaeffer

[9] Identities in the case of dynamic programming refers to obtained scores, in the case of the AAE algorithm, the cells in the table store 'shapes'.

1995), the number of sequences it is possible to align using this method in a reasonable timeframe is assumed to cut off at 7 (Durbin et al. 1998, p. 142; Gupta, Kececioglu & Schaeffer 1995).

2.3.2.2 Progressive Alignment:

The runtime cost of dynamic programming techniques encouraged researchers to approach the process of sequence alignment from different perspectives. Progressive alignment is a hybridized approach whereby multiple sequences are aligned incrementally, generally by continuous pair-wise alignments (Mount 2001, p. 152). Durbin et al.(1998, p. 144) suggest that there are three ways such progressive methods differ from one another; first of all, the initial logic they employ to order the sequences being compared. Second, whether the phylogenetic[10] tree that is created during processing involves a single alignment or several sub-related alignments. Finally, they may differ in the procedure and heuristics used to score sequences and alignments against one another. Despite the ability of such progressive systems to compute a significant number of sequences into a multiple alignment, Mount (2001, p. 155) cautions that the result is not guaranteed to be an *optimal* alignment. One example of a popular progressive alignment system is:
ClustalW: A system whereby sequences are progressively aligned in a pair-wise manner, with weighting selectively applied to sequences and substitution costs in order to create the most accurate alignment. Further explanation can be found in (Thompson, Julie D., Higgins & Gibson 1994).

2.3.2.3 Iterative Alignment:

The *iterative* approach to MSA attempts to address the problems inherent in the progressive alignment process; namely that the quality of the final MSA rests heavily on the quality of the initial alignment and the order of subsequent pair-wise alignments (Gotoh, O. 1996). As suggested by its name, such an approach repeatedly re-aligns regions and subgroups of sequences so as to provide a global alignment that is more optimal than would be produced had such optimization not been performed. One well-known iterative system is:

DIALIGN: A program conceived of by Morgenstern *inter alios* (1998) that attempts to create optimal global alignments via progressive alignment and then iterative weighting of local regions (or *motifs*[11]). According to the authors, DIALIGN contrasts itself from more traditional methods by ensuring that the local similarities between sequences are used to construct an MSA that displays an alignment that 'maximizes the sum of individual similarity scores' (Morgenstern et al. 1998, p. 293).

2.3.2.4 Consistency-based Alignment:

An element of most alignment strategies is the substitution matrix. As mentioned, often the measure of goodness for a particular alignment may be biased depending on the particular substitution matrix employed. Consistency-based alignment strategies may employ the methods used by the previously mentioned approaches (exact, progressive, and iterative) to construct pair-wise alignments. At this point, the similarity to these employed mechanisms diverges; the measure of goodness employed by consistency schemes defines the optimal MSA to be one that is most representative of all possible pair-wise alignments in a given set of sequences (Notredame 2002). Unfortunately, according to Kececioglu, computing such an optimal alignment is an NP complete problem (1983). One well known MSA application that employs the principles of consistency-based alignment is *T-Coffee*. The basis for a residue pair's score is given by what Notredame et al refer to as a 'position-specific scoring scheme'. Pair score is dictated by the compatibility of the given pair with the n sequences that comprise the library of weighted pair-wise alignments generated in the initial stages of T-Coffee's alignment process (Notredame, Higgins & Heringa 2000).

ap[11] Motif here refers to a 'sequence motif', that is, a recognizable pattern in a nucleotide or amino-acid sequence that is assumed to have some biological significance (Campbell, Reece & Meyers 2006).

2.4 THE ALIGNING ALIGNMENTS EXACTLY ALGORITHM:

The AAE algorithm as previously mentioned is an approach to MSA. It concerns in particular, the alignment of two multiple sequence alignments (Kececioglu, J & Starrett 2004). The algorithm is claimed by Kececioglu and Starrett to be capable of producing *optimal* alignments on benchmark instances in two widely used datasets. This claim might appear curious, given their complementary statement that aligning alignments is NP-complete, however they go on to emphasize that while the problem is inherently hard in the *worst* case, it is possible to produce a polynomial-time solution in practice. What is perhaps most interesting about AAE is that it claims to be able to produce *optimal* alignments, which are by their very definition, desirable to researchers.

A detailed explanation of the exact algorithm in the space available is infeasible, however the problem behind AAE is discussed in some detail in section 2.4.1 and a summary of the solution is provided in section 2.4.2. Initially however, it is instructive to consider some elements of the AAE algorithm itself.

First of all, the product of AAE is an optimal alignment of the two input MSAs, being able to handle instances with approximately 100 sequences consisting of 1000 columns (2004, p. 95). Second, these optimal alignments may be achieved using a technique Kececioglu and Starrett refer to as 'dominance pruning' (2004, p. 94) that allows for computation in linear space. Furthermore, the application of the proposed 'bound pruning' (2004, p. 95) technique, may decrease the time complexity by an order of magnitude while having the side effect of increasing space cost to quadratic. Finally, the exact algorithm has been rigorously tested on two independent benchmark datasets(2004, pp. 92-94); namely the BAliBASE and that of McClure, Vasi, and Fitch (McClure, Vasi & Fitch 1994). The results presented by Kececioglu and Starrett provide plausible substantiation that the AAE algorithm is effective in practice, despite its worst-case complexity. Let us now examine AAE in order that a methodology may be derived for investigating the hypothesis.

2.4.1 THE PROBLEM:

Kececioglu and Starrett (2004, p. 86) define the problem of 'Aligning Alignments' as:

"The Aligning Alignments Problem is the following. The input is a pair of multiple alignments A and B, weight function ω on pairs of strings from A and

B, substitution cost function δ, gap initiation cost γ, and gap extension cost λ. The output is an alignment of the columns of A versus the columns of B that minimizes the sum-of-pairs objective with linear gap costs."

This formidable definition contains several key elements:

- Alignments A and B are defined as being multiple alignments if and only if any collection of strings S which may be composed into an alignment contains more than two strings.
- Any string in alignments A and B (therefore, from collection S) may contain only characters from a given alphabet Σ or a spacer (gap) character, which is denoted as '-'.
- An assumption is made that such gaps may run for $\geq$ one column. Hence, the cost for any given gap of length x is defined as $(\gamma + \lambda x)$ where γ is a constant ≥ 0 that is defined as the cost of initiating a gap, and $\lambda \geq 0$ is the cost of extending a gap.
- Given that substitutions will undoubtedly occur, it is necessary to define a substitution cost function δ that assigns each pair of letters a, b the cost $\delta(a, b) = \delta(b, a)$. Therefore, define alignment cost f as the sum of all substitution costs $\delta(a, b)$ over all non-gap columns $\begin{matrix} a \\ b \end{matrix}$, plus the sum of the gap costs.
- The *sum-of-pairs objective* scores a given multiple alignment A in the following manner. Given two rows i, j in alignment A, induce a pair-wise alignment A_{ij} of the strings S_i and S_j. A weight function ω is defined to assign each pair of strings S_i, S_j the weight $\omega(i, j) = \omega(j, i)$.

According to the authors, therefore, the *sum-of-pairs cost* of alignment A is defined to be the weighted sum of the costs under cost function f of the two string alignments induced by all unordered pairs of rows. The cost of A_{ij} is weighted by $w(i,j)$. Hence, the pair (ω, f) specifies the sum-of-pairs objective function (Kececioglu, J & Starrett 2004).

2.4.2 The exact algorithm (Summary):

The AAE algorithm essentially follows the dynamic programming method as originally presented by Needleman and Wunsch (1970). This is with respect to its representation, as we view the alignments as a series of columns. Therefore, in order to align an alignment A of k rows and m columns to an alignment B of l rows and n, AAE requires the construction of a grid-structured graph of dimensions $m + 1$ by $n + 1$. Next, the graph is examined in lexicographic order, which corresponds to traversing the graph in row-major order until the cell (m, n) has been calculated.

To determine the cost of any alignment the exact algorithm produces, it is necessary to establish the number of gaps initiated by a given column. The proposed theory for doing this rests on the concept of a shape. Shapes are ordered partitions of the rows of both alignments, indicating the ordering of each row's final character. This is most easily explained with an example; consider the shape $\{(4)(2), \{1,3\}\} -$ each number in this construct represents a pair of rows in a given alignment A such that row 4 finishes first, with its last character followed by gaps, row 2 finishes second, while rows 1 and 3 finish after both 4 and 2, however in this particular shape, rows 1 and 3 end in the same character (either a letter or a gap) and as such are called 'flush'. The algorithm dictates in this circumstance that row 4 *underhangs* rows 2, 1 and 3, and row 2 *overhangs* row 4 respectively. Because there may be multiple alignments generated during the alignment process, each cell in the dynamic programming table maintains a list of shapes and scores for the optimal alignment that ends in that particular shape.

Given the ability to generate shapes and the multiple alignments A and B that were of sizes $(k \times m)$ and $(l \times n)$ respectively, the optimal alignment of A and B is generated by solving the following subproblem: for any shape s and indices $0 \leq i \leq m$ and $0 \leq j \leq n$, the cost of an optimal alignment of the prefixes $A[1{:}i]$ and $B[1{:}j]$ that ends in shape s. The authors call this solution cost $C(i, j, s)$. Unsurprisingly, it is not possible to compose every shape s by aligning all given prefixes. . Hence, set of all possible alignments given these prefixes is denoted as $S(i,j)$ for a particular cell in the dynamic programming table. The cost of an optimal alignment of A and B is defined to be:

$$\min_{s \in S(m,n)} \{C(m,n,s)\}.$$

In order to count gaps the following predicates are employed:

For a pair of rows p and q in an alignment with shape s.

${}_q s^p$ if and only if p overhangs q in the alignment, and

${}^p s_q$ if and only if p underhangs q.

And for the rows p and q and a column c,

${}_q c^p$ if and only if p has a letter and q has a spacer in column c, and

${}^p c_q$ if and only if q has a letter and p has a spacer.

Thus when aligning two columns a and b, the composite column is formed by placing a on b. Using this notation, a and b will columns from A or B or alternatively, a column of all spaces (gaps). So, the total number of gaps that are initiated by appending column (a,b) onto any new alignment that ends in shape s is:

$$g(a,b,s) = \sum_{\substack{p \in A \\ q \in B}} \left(\left({}_q(a,b)^p \ and \ !_q \, s^p\right) \ or \ \left({}^p(a,b)_q \ and \ !^p \, s_q\right)\right).$$

Assume in this equation that any predicate that evaluates to true is equal to 1, and the opposite evaluates to 0. A 'flat' shape is defined to be a shape in which all rows are flush, where the associated alignment concludes with a column composed of all letters, or is empty altogether:

$$\varphi = (\{1, \dots, k+l\}).$$

Therefore, to determine the shapes within any given cell (i,j), the following recurrence is established:

$$S(i,j) = \begin{cases} \{\}, & i < 0 \; or \; j < 0; \\ \{\varphi\}, & i < 0 \; and \; j = 0; \\ S(i-1,j) \; o \; (A[i], -) & \\ \bigcup S(i,j-1) \; o \; (-, B[j]) & \\ \bigcup S(i-1,j-1) \; o \; (A[i], B[j]), & otherwise. \end{cases}$$

Where $S(i,j)$ is the set of all shapes at any given cell in the table, and given a column c, $s \; o \; c$ is the shape derived from concatenating c onto an alignment ending in shape s. Therefore, $S(i,j) \; o \; c$ is the set of shapes derived from concatenating c onto any s such that $s \in S(i,j)$. Assume in this circumstance also, that any $A[i]$ refers to any column i in alignment A, and $B[j]$ refers to any given column in alignment B. Once $S(i,j)$ is established, the sub-problem of $C(i,j,s)$ must be solved. The recurrence for $C(i,j,s)$ relies on the assumption that the optimal alignment of $A[1{:}i]$ and $B[1{:}j]$ ending in shape s must have the final column c such that with the removal of c an optimal alignment remains that ends in shape š where $š \; o \; c \; = \; s$. Hence assuming $0 <= i <= m$ and $0 <= j <= n$, and $t \in S(i,j)$, and $(i,j) \; != \; (0,0)$:

$$C(i,j,t) \;=\; \min \begin{cases} \min\limits_{\substack{s \in S(i-1,j) \\ s \,\circ\, (A[i],-) = t}} \begin{Bmatrix} C(i-1,j,s) \\ + \gamma \, g\big(A[i], -, s\big) \\ + \lambda \ell \, \big|A[i]\big| \end{Bmatrix}, \\ \min\limits_{\substack{s \in S(i,j-1) \\ s \,\circ\, (-,B[j]) = t}} \begin{Bmatrix} C(i,j-1,s) \\ + \gamma \, g\big(-, B[j], s\big) \\ + \lambda k \, \big|B[j]\big| \end{Bmatrix}, \\ \min\limits_{\substack{s \in S(i-1,j-1) \\ s \,\circ\, (A[i],B[j]) = t}} \begin{Bmatrix} C(i-1,j-1,s) \\ + \gamma \, g\big(A[i], B[j], s\big) \\ + \sum\limits_{\substack{p \in A \\ q \in B}} \delta\big(A[p,i], B[q,j]\big) \end{Bmatrix}, \end{cases}$$

Where $|c|$ denotes the number of letters in column c. Therefore, for $(i,j) = (0,0)$,

$$C(0,0,\varphi) := 0.$$

2.4.3 Benchmark Test Suites:

For the purposes of this research, test data will be instrumental in determining whether the objective has been achieved and has satisfied the demands of the hypothesis. Given the problem domain, it would be most instructive to utilize the test data that was used to determine the performance of the exact algorithm in the first place. As such, any experimental system will be tested on the BAliBASE test suite. BAliBASE is specifically designed to allow for the benchmarking of MSA algorithms (Thompson, J. D., Plewniak & Poch 1999) and has evolved considerably since its inception (Julie D. Thompson 2005). The referenced papers provide an excellent overview on the constituent elements of BAliBASE and its adoption rate since it was published in 1998. One other documented set of test data that Kececioglu and Starrett tested AAE on is that of McClure, Vasi, and Fitch (MVF) (1994, p. 573). While the MVF dataset was carefully selected to provide a fair analysis of MSA applications, the difficulty of obtaining the original dataset and its age in comparison to the BAliBASE (given that BAliBASE has been maintained and improved upon since its inception) renders the MVF to be a less attractive candidate for use as test data. The architects of BAliBASE (Thompson, J. D., Plewniak & Poch 1999) clarify the need for distinct databases of accurate reference alignments for the use of testing MSA programs given that MSA algorithms by their very nature require more than two sequences to produce a compelling alignment. They stipulate that the performance of an alignment program rests upon 'the number of sequences, the degree of similarity between sequences and the number of insertions in the alignment. Other factors may also affect alignment quality, such as the length of the sequences, the existence of large insertions…' (1999, p. 87).

2.5 Existing applications of parallel sequence alignment:

Given the hypothesis, it is instructive to examine previous examples of parallelised sequence alignment methods. Perhaps unsurprisingly, most examples of parallel sequence alignment (at least, in the area of MSA) attempt to parallelise either progressive or iterative alignment algorithms (Ebedes & Datta 2004; Li 2003; Schmollinger et al. 2004; Yap, Frieder & Martino 1998). What is perhaps more useful however, is to consider how the dynamic programming approach has been parallelised, given that it is an integral part of the AAE algorithm. Typical parallelization methods for the dynamic programming approach are usually concerned with applying the approach to comparing a single sequence against N sequences (Brutlag et al. 1993; Trelles-Salazar, Zapata & Carazo 1994);this is due to the practical concerns of searching databases for particular sequence similarities. Parallelization of the actual algorithm itself (and not the large-scale *application* of the algorithm) is a less well-studied field. Martins et al. (2001)discuss a method whereby it is possible to parallelise the composition of the dynamic programming table. They suggest three possible methods of parallelising this particular step in the algorithm. In particular, given that the value of any cell δ is dependent on its north, north-west and western neighbour cells:

Parallelise the computation by allocating a processor to each row, using discrete time intervals to co-ordinate communication.

Perform a similar operation, only on the columns of the table rather than the rows.

Or, anti-diagonal by anti-diagonal, allowing the computation to progress in the original top-down method.

The initial two approaches are discarded by Martins et al. due to the fact that elements within a given row or column depend on other elements within that same row or column. Hence, it is not possible to parallelise the composition of individual rows or columns (Martins et al. 2001). The third approach minimises this problem in that any diagonal d is dependent on only the three neighbours as mentioned above. Problems inherent in this approach are communication costs; all processing units would need to communicate among one another in a somewhat complicated fashion. Also, assuming that each processor calculates only one value in the table, it requires three input values and creates only one output value; a $3:1$ communication to computation cost. Martins et al. (2001) circumvent this overhead by performing a block division upon the table whereby a certain range of elements are assigned to a given processor to be computed at any single time step. Dependencies still exist with this method, however communication overhead is reduced: assume each block contains 4×4 rows and columns respectively – said block requires 9 input values (those to the west of the block, those north, and a single element in the north-west), yet computes 16 minima, dropping the communication to computation ratio to $9:16$. Further improvements are also mentioned, namely the horizontal striping of the block table in order to alleviate processor load balancing issues. Martins' approach bears particular relevance to our own hypothesis, given that one of his aims was performance enhancement on general purpose parallel computing platforms.

2.6 Summary:

This review has presented an introduction to the relevant issues pertaining to the investigation of the stated hypothesis. An attempt was made to relate the field of sequence alignment theory to the issues inherent in the parallelization of applications. A parallel version of the Needleman-Wunsch algorithm was examined that may provide some insight into the investigation of the current hypothesis. Naturally, given the scope of the review it is impossible to explain exhaustively all relevant issues, though attempts have been made to direct curious readers towards the appropriate resources. At each stage it was necessary to consider the utility of a particular subject area given the hypothesis. Subsequent chapters discuss the method employed for the parallelization of the AAE algorithm and what results have been gathered.

3. Approaches to Parallelization

This chapter deals with the approach taken to parallelize the AAE algorithm. Initially, it describes the issues with the algorithm that prevent it from being *embarrassingly parallel* and how these may be alleviated to successfully parallelize one particularly time consuming operation. The choice of this particular operation as a suitable candidate for parallelization is discussed in the second section, as is the motivation for using the dominance pruned variant of the algorithm during this research.

3.1 Considered elements

3.1.1 Dynamic programming table generation

Kececioglu and Starrett (2004) describe the initial stage of the AAE algorithm as using a dynamic programming table. This matrix is the same grid-structured graph described in section 2.4.2 and is composed over the columns of any two alignments A and B. For each entry i,j in the table a list of *shapes* is maintained, the generation of which comprises the initial stage of the algorithm. Given that in the worst case the number of shapes at an entry is exponential in k and l (where k and l are the number of sequences in the alignments A and B) the problem of computing this matrix is not trivial.

Another issue worth considering with regard to any useful parallelization of this particular algorithm are the inherent dependencies present. As mentioned in section 2.5, Martins et al (2001) had some success with their parallelization of the computation of a dynamic programming matrix. They observed that any potential speedup factors would be confounded by the dependencies present in the underlying algorithm; dependencies which exist through generalisation of the dynamic programming approach, in the AAE algorithm. This corresponds to the necessity of computing the shape-list of neighbouring cells to the north, north-west and west of any cell c before the computation of c may commence. Figure 2 demonstrates this relationship, with each arrow corresponding to a required propagation of a shape. In the context of the exact algorithm, the relative finishing order of each arrow in the figure dictates the order in which each shape from the cell i,j is propagated. This order can be important when deciding whether to maintain a shape in a cell's shape list, especially when incorporating a pruning technique into the algorithm.

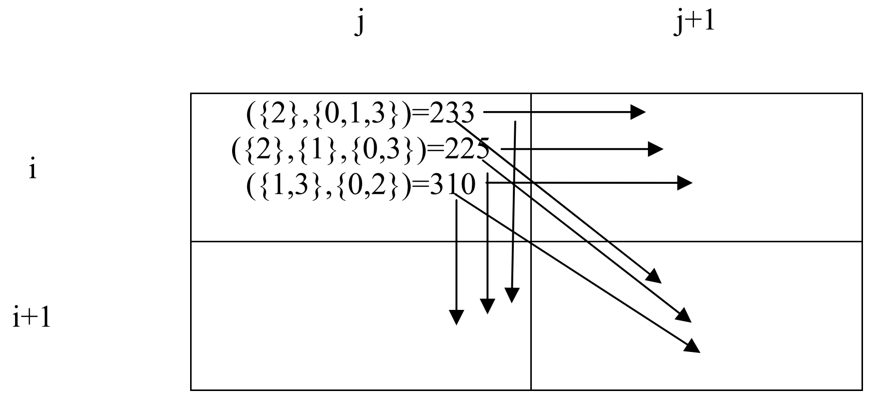

Figure 2: Shape propagation in the dynamic programming matrix

Section 2.4 mentions *dominance* and *bounds pruning* techniques as allowing a substantial reduction in the time and space complexity of the AAE algorithm (in particular, the generation of the dynamic programming matrix); therefore, the theory of these techniques is discussed in the following section.

3.1.2 DOMINANCE AND BOUNDS PRUNING

In order to reduce the time and space complexities of the AAE algorithm and render it useful as a tool for molecular sequence alignment, it was necessary for the authors of the algorithm to conceive of a method whereby unnecessary operations were prevented from occurring. Kececioglu and Starrett (2004) propose three techniques for improving the performance of AAE.

3.1.2.1 Space reduction under Hirschberg's principle

The maximal common subsequence theory discovered by Hirschberg (1975) dictates that it is possible to compute an optimal alignment of any two strings a and b in space linear in the lengths of a and b respectively. Myers and Miller (1988) go further, discussing a generalisation of Hirschberg's theory to allow the alignment of two strings with linear gap costs; an essential step in some forms of molecular sequence alignment. Furthermore, it is possible to generalise this divide-and-conquer approach to the problem of aligning alignments. Kececioglu and Starrett (2004) describe a method whereby the optimal alignment is produced via a row-swapping approach made possible by the dominance pruning heuristic (see next section). While this has the effect of reducing the space consumption of AAE to linear in the number of columns of the input alignments, it does not improve the runtime significantly; time complexity remains quadratic in the number of shapes generated during the row-swapping procedure. Nevertheless, this is an attractive mechanism that could be employed if it were necessary to use the algorithm on a workstation with limited memory.

3.1.2.2 Dominance Pruning

Reducing the space of AAE is an imperative to its successful conclusion; dominance pruning allows for significant space saving while having the side-effect of improving the timeliness of the algorithm markedly. Essentially; establish a dominance relation that operates on pairs of shapes. As mentioned, each cell in the dynamic programming matrix represents a list of shapes. Given some cell C with associated shapelist L the total number of shapes maintained by the vanilla algorithm is exponential in the number of sequences, hence L^{kl} where k represents the rows from some alignment A and l represents the number of rows from some alignment B. This is clearly unacceptable on all but the most trivial of inputs. Therefore, by establishing a recurring dominance relation on pairs of shapes, it is possible to interleave the propagation and pruning of shapes in such a manner that the resulting shape list is equivalent to its form if it had been pruned post-propagation. This is established by the following logic. At each stage in the algorithm, from some location i, j in the dynamic programming matrix, three operations can be completed; an insertion, a deletion, or a substitution $-$ note that these operations are those of propagating a shape to the east, south or south east of the current cell. Each of these operations yields a subalignment of the current alignment that Kececioglu and Starrett (2004) term an *extension*. The dominance pruning technique seeks to establish a relation where for shape t in some list L, remove t if it is never better than another shape s from the same list over all possible extensions from that cell. Thus, for shape s and extension ρ let $so\rho$ denote an alignment derived from composing ρ onto the pre-computed alignment associated with s. Let $C(s)$ denote the cost of an alignment ending in shape s, and $C(\rho)$ denote the cost of subalignment ρ that begins with the flat shape (the initial shape in the matrix $-$ wherein all sequences finish at the same point and associated cost is 0). Further, let $G(s,\rho)$ represent the count of the numbers of pairs of rows $p \in A$ and $q \in B$ such that p overhangs or underhangs q in a gap that is continued by ρ. It is possible then to establish:

$$C(so\rho) = C(s) + C(\rho) - \gamma G(s,\rho).$$

Where γ represents the gap initiation cost discussed in 2.4.1. To extend this, shape t is no better than shape s on any extension ρ if it is possible to show the following:

$$\begin{aligned}
&C(to\rho) - C(so\rho) \\
&= \big(C(t) - C(s)\big) - \gamma\big(G(t,\rho) - G(s,\rho)\big) \\
&\geq \big(C(t) - C(s)\big) - \sum_{\substack{p\in A \\ q\in B}} \big(({}_q t^p \wedge !_q s^p) \vee ({}^p t_q \wedge !^p s_q)\big) \\
&\qquad \geq 0,
\end{aligned}$$

Wherein the definition may be simplified because the resulting upper bound on pairs of rows that satisfy the overhang/underhang conditions is independent of ρ. Therefore, *shape* s *dominates shape* t *if:*

$$C(t) \geq C(s) + \gamma \sum_{\substack{p \in A \\ q \in B}} \left(({}_q t^p \wedge !_q s^p) \vee ({}^p t_q \wedge !^p s_q) \right)$$

This definition is of particular importance given that it means that in the *best* case, the shape-list at a given cell can be pruned to length 1. Note the two fold benefits of this operation; while it requires some overhead to perform the dominance calculation $-$ it will result in less work over time given that there will be fewer shapes to propagate, which in turn, dictates that there will be a lower storage overhead. Furthermore, it is possible to combine dominance pruning with the row swapping space reduction technique mentioned earlier. This is the method by which the algorithm may run in space linear in the number of input columns from any two alignments A and B.

3.1.2.3 Bound Pruning (an overview)

Bound pruning seeks to improve the time complexity of the exact algorithm by computing upper and lower bounds U, L on the cost of the 'best alignment of A and B that extends the alignment associated with shape s' (Kececioglu, J & Starrett 2004). Unlike dominance pruning, bound pruning cannot be used alongside the row swapping technique, rendering its space complexity to be $O(n^2)$ where n is the number of input columns from alignments A and B. One of the benefits of bound pruning, however, is that unlike dominance pruning, in the *best* case, the algorithm can prune an entire shape-list from the table, thereby decreasing the amount of work that must be performed during composition. Unfortunately, bound pruning requires the computation of three tables for optimistic scores; one for insertions, one for deletions, and a final table for substitutions (Kececioglu, J & Starrett 2004). This has the net effect of imposing a great deal of overhead in situations where it would be desirable to subdivide a task for parallel processing. These storage issues result in an already highly dependent process (the generation of the dynamic programming matrix) having further dependencies introduced that render a bound-pruned matrix infeasible to compute over several machines.

3.2 Logic & Pragmatism

This section examines the logic behind the design decisions made for the parallel implementation of the AAE algorithm. First is an overview of the approach used for subdividing the matrix into distinct entities that may be assigned to individual processing units; second, justification is provided for the appropriateness of the chosen parallelization scheme.

3.2.1 Subdividing the Dynamic Programming Matrix

As has been previously mentioned, the most expensive aspect of the exact algorithm is that of composing a matrix of *shapes*. The time and memory complexity may be improved upon using bound and dominance pruning, however, as stated in the hypothesis; for large alignments it may be possible to improve upon the speed of the sequential algorithm by parallelizing the composition of the shape matrix. The approach taken to improve timeliness has two aspects.

3.2.1.1 Blocking as an efficiency enhancer

The first aspect is that of dividing the elements of the dynamic programming table into *blocks*. A block is defined as a subset of cells from the dynamic programming table, such that $b \in \zeta$ where $b \subset C$. C denotes the set of all cells in some matrix M, and ζ denotes a stripe from the set of stripes Z. Stripes are defined and discussed in the following section (3.2.1.2). The following sections also discuss how blocking is used practically to improve efficiency by aggregating work for processing.

3.2.1.2 Striping to maximise productivity

The second aspect to improving the timeliness of the exact algorithm is that of block striping. In the context of the parallel exact algorithm, a stripe may be defined as a subset of the rows in some matrix M. Therefore, given the set of all rows R, define a set of stripes Z such that each stripe $\zeta \in \mathrm{Z}$ and $\zeta \subset R$. As discussed in Martins et al (2001), it is advantageous to ensure that all processing units (hereafter referred to as pUnits) are always assigned some kind of work (Martins refers to this concept as 'strips'). Stripes are one mechanism whereby it is possible to maximise processor productivity. Conceptually, assigning a stripe to a pUnit has the effect of making that pUnit responsible for all blocks in the stripe ζ. The utility of this approach is discussed next.

3.2.1.3 Using stripes and blocks to improve timeliness

Given the definitions of blocks and stripes provided, the question remains how can they be used to practically improve the timeliness of the exact algorithm? Wilkinson and Allen's (1998, p. 26) *requisite communication versus expected computation* ratio (as defined in 2.2.2.2) dictates that in distributed parallel applications, it is beneficial to aim for a ratio that is biased towards a higher expected computation outcome. Essentially, the algorithm should achieve a processing output that is greater than the required communication for that particular output. In terms of the parallel exact algorithm; both striping and blocking help lower the communication to computation ratio significantly. Recall that each cell in the dynamic programming matrix is dependent on its *northern, north-western,* and *western* neighbours for input. Furthermore, these neighbours must be complete in the sense that their shape-lists must be their final length; this is to say, an assumption is made that the shape-list in question is now immutable to the degree that no further write operations will be performed upon it.

A naïve parallel implementation might consider allocating the computation of individual cells to distinct pUnits. The reasons for avoiding such a method are immediately obvious; for each cell, it is necessary to retrieve the neighbouring cell's final shape-lists. This would result in a communication to computation ratio of $3{:}1$, with three communications required to process a single cell. To extend this approach, it is equally possible to assign a pUnit to an individual row (or column) in the dynamic programming matrix. This is a similar method to striping, but is still less effective than desirable – the computation of a single row may not be a very costly exercise: presuming the pUnit cached the *western* neighbour cell's values (assuming it processed the western cell in a *row*), the communication ratio inherent in this approach would be $2:1$ with the cell to be computed being reliant on the *northern* and *north-western* shape-lists.

It seems logical, therefore, that improving the timeliness of the exact algorithm is dependent on reducing the amount of communication that is inherent in any parallel implementation. Blocking is the primary approach used for improving the ratio of communications versus work undertaken by the remote-hosts[12]. The definition of a block in section 3.2.1.1 defines a region of the underlying matrix for computation. Varying the logical horizontal and vertical measures of a block will result in the following results: by increasing the block size, more work will be undertaken versus the communication required to initiate the job; conversely, the inverse effect results from decreasing block size. Consider the following example; given some block B of size 5×5, presume that input is *only* required to be transmitted between pUnits for the first row in B. This relies on several assumptions; first, that the pUnit in charge of this block has cached the results of the *western* block if indeed there was a *western* block: if B can be identified as being from the initial column of blocks in the logical matrix that is the set of all stripes Z, there will be no input from the western block. Second, that the communicable pre-requisite for computing B is the final row of the block to the north of B. If these conditions hold, the communication to computation ratio for B would be $5:25$, such that for 5 communications of cells, 25 cells are computed.

0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19
4	20	21	22	23	24

Figure 3: A example of a 5x5 block with numbered cells

Clearly, this is preferable to a ratio of $3:1$ for the naïve parallel approach; but as it requires a pUnit to store the results of the final

[12] The implementation of the parallel algorithm was that of a distributed process; see section 3.2.1.5.

column for the previously computed block ($B-1$) in all cases except for the first block in a row of blocks r_b, further logic is necessary to achieve this particular ratio. Note that if the cached column was not available, the ratio would be increased to $11{:}25$, which is a result of having to communicate the details of the previous block $B-1$ to the current block B. It would also be necessary to communicate the value of the final cell of the block immediately to the north-west of B. Striping is the requisite logic that allows for the caching of a column to serve as input to the next block. This logic sounds slightly off, but is redeemable if the cached column is of size B_h+1, where B_h denotes the height of an individual block in *cells*.

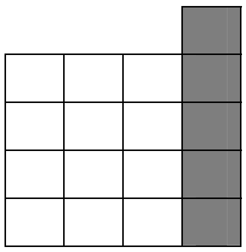
Figure 4: A 4x4 block with logical cached column highlighted[13]

It is important to note that when a pUnit processes a particular stripe, any block B is *always* dependent on its neighbouring block $B-1$. If this ordering is violated, the produced alignment will be incorrect, or in the worst case, the program cannot run to completion. The use of stripes as a mechanism for improving timeliness rests on the concept that given x stripes over some table T and the set of pUnits j, each pUnit $p \in j$ will be allocated some stripe s_i where the range of i is 0 to $x-1$. In instances where the number of stripes x is the size of j, this mechanism is trivial: each p receives its own s_i. Logically, this means in terms of computation that in all cases excepting where $i=0$ (for the first stripe to be computed) that p is dependent on input from $p-1$, which

[13] The implementation does not make any attempt to store such a peculiar data structure; rather, it stores a 1 dimensional representation of the column between jobs.

corresponds to s_{i-1} (the previous stripe). The dependency in this case is a block-level dependency; if a pUnit were required to wait for the composition of an entire stripe before performing its own tasks, there would be a surfeit of available processing time that would be wasted. Hence the segmentation of a stripe into blocks; at any stage, a pUnit undertaking the composition of some block B in some stripe s_i requires the final row r_n of the preceding block vertically in the table, $Bs_{i-1}r_n$.

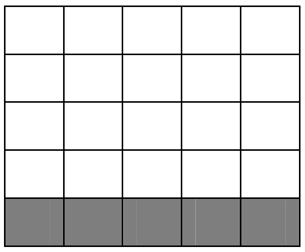
Figure 5: Input row from a 5x5 block

This means that any p is always going to be dependent on s_{i-1} which suggests that there is a threshold upon which no further speedup can be gained, a threshold that is logically based upon the processing of blocks in a stripe, and physically based upon the maximum possible rate at which a pUnit can complete a block. The speedup for the parallel algorithm is discussed in the following section.

3.2.1.4 Speedup of the exact algorithm.

As stated in section 2.2.2.2, the speedup factor for a particular parallel algorithm over its sequential counterpart is formalised thus;

$$S = \frac{\alpha}{\beta}$$

Where there is a tacit assumption that given a maximum factor N, it is always the case that $S \neq N$, but that in certain circumstances $S \approx N$. The results section details the *actual* results that were achieved through experimentation on different types of input data. Here, the theoretical speedup of the parallel algorithm is described. Therefore, let C_s denote the sequential computation time over some table T and C_p denote the parallel computation time over T. It is assumed that:

$$C_s \neq C_p$$

And that,

$$C_s \wedge C_p \propto \theta_T.$$

Such that θ_T denotes the *number of cells* in T. Section 2.4.2 describes how θ_T can be calculated; essentially, it is the product of the number of characters in a sequence from both alignment A and alignment B. More simply assuming a denotes a sequence from alignment A and b denotes a sequence from alignment B:

$$\theta_T = L(A_a) \times L(B_b).$$

Where $L(x)$ determines the number of characters in a given sequence. This reasoning follows the original definition of the exact algorithm in Kececioglu and Starrett's (2004) paper in that the complexity of generation of the dynamic programming table is a function of the number of *columns* in the input alignments A and B, rather than the number of sequences in each alignment. To calculate the speedup for the parallel algorithm, there are two factors that must be considered: first, the sum of the communication penalties cp_i (see section 2.2.2.2) on all processors:

$$tcp = \sum_{i'=0\ldots q} cp_{i'}$$

Where tcp represents the total communication penalty over all pUnits, i' denotes a logical pUnit from the set of pUnits j and has the range 0 to q where q is the size of j. Within the expression of cp_i is the term C_i which denotes the time of communication between processing units. To calculate C_i, it is necessary to introduce the second factor that must be considered in the parallel speedup; a wait-factor w that scales linearly with q. This wait-factor is a function of several elements of the parallel algorithm; the processing of blocks, the number of pUnits q, the number of blocks ς in a stripe, and the number of stripes in the table δ. Therefore, let B_t denote the time to compute a single block;

$$\zeta_t = \sum_{i=0}^{\varsigma} B_t^i$$

Where ζ_t denotes the time to compute some stripe ζ in table T. Recall that the parallel algorithm attempts to maximise productivity by assigning stripes to pUnits. Given that it proceeds to compose the table in row-major order (over all stripes), it may be the case that there are more stripes in the stripe set Z than there are in the pUnit set j; this results in the necessity of cycling pUnits: once a pUnit finishes a stripe, it will be moved to the next unassigned stripe in the table, if any remain to be processed.

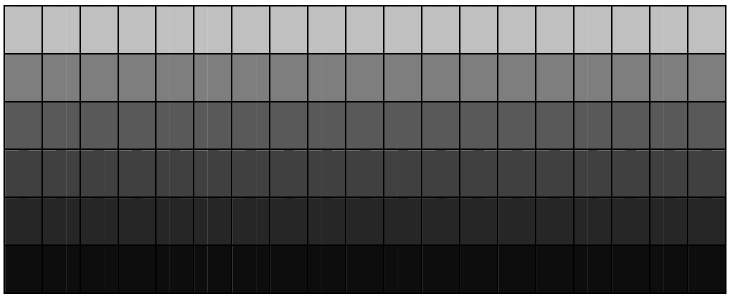
Figure 6: 6 stripe by 19 block overlay matrix. Darker shades indicate longer waits for jobs

The cycling of stripes and the dependency between stripes introduces the last aspect of the wait factor; time spent waiting for vertical communications down the overlay matrix[14]. Thus, each pUnit $p^{i'} \in j$ (where i' denotes an index of j) must idle for time proportional to the number of pUnits processing jobs before it. This allows for the following definition:

$$\Upsilon = \sum_{i=0}^{i\prime} B_t$$

Where Υ denotes the time spent idling while waiting for the preceding stripes to be computed for any pUnit $p^{i'}$. The product of the sum of all Υ over T and the value of ς provides us with the final calculation for all C_i. Maximum communication cost over all processors is defined as:

$$C = \sum C_i$$

[14] Blocks and stripes together form a matrix that logically ‘overlays’ the existing dynamic programming matrix.

Assuming,

$$C_i = \varsigma \times \Upsilon_T^i$$

Where C is the sum of all C_i and Υ_T denotes the sum of all Υ over T. It is assumed that ς denotes the number of blocks in a stripe ζ. Thus, the speedup factor for the parallel exact algorithm may in fact be *limited* when a large number of pUnits is present; given the size of Υ in those circumstances. It would be illuminating to conduct tests of the parallel algorithm over a large number of pUnits to determine if this is the case practically, however this is out of the scope of the current research given limited resources. The results section details actual speedup factors achieved in varying circumstances.

3.2.1.5 Parallel exact algorithm walkthrough

The following section provides an abridged walkthrough of the parallel algorithm as described to some extent in the previous sections. It is not the intention of this section to provide an exhaustive explanation of particular implementation details, but rather to allow the reader to understand key aspects of the algorithm and how they can be implemented in code. The intention of the parallel exact algorithm is to *subdivide and process in parallel the programming matrix construction phase* of the exact algorithm. As such, it begins with two alignments, A and B that are presumed to be in the legal form[15] of input alignments to the original exact algorithm. A dynamic programming matrix of the requisite size is constructed and importantly, the *first* row is filled with shapes by way of pre-processing. Hence the algorithm may be summarised in pseudo-code thus:

```
Pre-populate initial matrix row;
numberStripes = (matrix numRows – 1) / blockSizeVertical;
Assign each pUnit a stripe;
While stripes are left to process:
Send input row for a block to assoc. pUnit;
pUnit computes block and returns;
```

[15] See (Kececioglu, J & Starrett 2004) for the definition of legal input.

```
Block is inserted back into final matrix;
Prepare new job based on returned block;
If pUnit finishes a stripe, assoc. it with a new stripe.
Conduct traceback serially;
```

Where at each point there are significant issues that must be addressed with regards to completing each step efficiently. The implemented version of the parallel algorithm was written as a distributed process. This was conceived of as being of more utility in a laboratory environment with several limited-power systems, rather than a localised parallel version that would require a significantly powerful machine to maximise its performance[16]. Where explanation of technical aspects is desired, consult the source code contained in the appendices. The method section of this dissertation provides an example of the network structure used during experimentation. What is of vital importance to understand when implementing the parallel algorithm is that the processing order of the algorithm must be preserved lest the resulting alignment be incorrect. More simply, it is impossible to perform certain actions before others; as explained earlier with regards to dependencies – the exact algorithm is a fundamentally serial process, with the computation of each cell being wholly reliant on the correct treatment of its neighbours.

4. METHOD

This chapter focuses on the procedure undertaken to obtain results and the metrics used to determine the quality of those results. Furthermore, an overview of the system architecture used for testing is provided, along with specifications on the machines used.

4.1 INTRODUCTION

The implementation of the parallel exact algorithm was targeted towards multiple systems in a miniature cluster. As such, efficient network communication and the careful divorce of worker and listener threads is required in order to allow for the successful completion of tests. The following sections describe the metrics used to evaluate the performance of the exact algorithm, a description of the dataset used

[16] Section 7 discusses the implications of employing the parallel algorithm on such hardware.

during experiments, and an overview of the system architecture with general assumptions on how the system should perform in certain circumstances. The results chapter details the outcome of the experiments mentioned here, along with speculation about certain aspects of the results and how they may have occurred.

4.2 METRICS

The primary metric for evaluating the performance of the parallel exact algorithm is that of time. More specifically, the time required to compute a complete dynamic programming matrix. It would be interesting to calculate the mean processing time for a particular block size when provided to the parallel algorithm; however this is unlikely to be useful especially given that the performance would be identical to that of the sequential exact algorithm working over a block of the same dimensions. The experiments undertaken focus on the composite performance of the algorithm on alignments of varying sizes rather than the performance of its constituent elements. Given further time it would be instructive to consider the capabilities of each element of the system employed for processing the algorithm so as to more accurately tune its performance.

In order to accurately measure the time of the algorithm, a timer was required. Initially, the `Stopwatch` class from the Microsoft .NET Framework 2.0[17] (2007b) was considered to fulfil this purpose; however, after experimentation it was revealed that the non-deterministic nature of certain aspects of the parallel algorithm results in timing inaccuracies when using this construct. It is possible to negate most of these ill effects by specifying a processor affinity for the thread undertaking timing duties, however this was not deemed worthwhile. The one benefit of forcing a thread to execute on a single logical pUnit is that of reduced migration cost. When a thread migrates from pUnit to pUnit, the cache for that pUnit must be reloaded, resulting in a non-trivial slowdown. The solution to avoiding this sort of complexity and 'second-guessing' of the operating system's scheduler is to perform the mathematics of the elapsed time in code. This translates to obtaining a system time stamp when the algorithm commences, and subtracting this value from a time stamp obtained at the conclusion of the run. More simply:

$$e_t = f_t - s_t$$

[17] The parallel distributed exact algorithm was written entirely in C# under Microsoft Visual Studio 2005©. To allow fair comparison, the sequential algorithm was also implemented in C#.

Where e_t is the total time elapsed, f_t is the finishing time for the run, and s_t is the start time respectively. Note that this approach is not suitable for circumstances where high precision is needed; the mechanism for retrieving the current time stamp in .NET 2.0 is the `DateTime.Now` property, which limits the measurement precision to milliseconds. Conversely, the `Stopwatch` class allows for microsecond precision while introducing the possibility of inaccuracies when used in a multi-threaded environment, as mentioned earlier. Given that the intention of this research is the evaluation of the run-time of the parallel algorithm over large instances, millisecond accuracy does not appear to detract from the final results.

In order to evaluate the performance of the algorithm, run-time performance in fractional minutes and seconds is recorded. The maintenance of these values is not useful without having an independent variable in the process; hence, the number of cells in the computed matrix and the dimensions of the alignments that contributed to the size of the matrix are also stored for analysis. The following section discusses the tests performed, the architecture of the system employed for testing, and assumptions that were inherent in the testing process.

4.3 Experiment Design

4.3.1 System Architecture

To evaluate the performance of the parallel exact algorithm, a micro-cluster of 6 machines was constructed. In addition to this 'slave-cluster', a slightly more powerful system was drafted to run as the master. Specifications of the machines used are as follows. Each slave node ran an Athlon 64 3200+ CPU with a default clock of 2.0GHz. Memory for each system was composed of 1GB of DDR RAM clocked at 400MHz. This was configured in two banks, allowing for dual-channel access rates[18] to improve performance. The master node ran an Intel Core 2 Duo T5500. Each core of this system was clocked at 1.66GHz with accompanying 1GB of DDR2 at 533MHz. While the master had a lower clock rate than the slave nodes did, this was offset by its ability to perform two tasks in parallel; an important factor in the parallel algorithm. Tests for the sequential algorithm were performed on a single Athlon 64 3200+, a system that would be incorporated into the micro-cluster for parallel testing.

[18] Dual-Channel in this sense refers to the ability to double data throughput from RAM to the memory controller (Kingston 2003) and therefore, logically, the CPU itself. Practically, this means the data path is 128 bits wide rather than 64, resulting in approximately double the possible bandwidth.

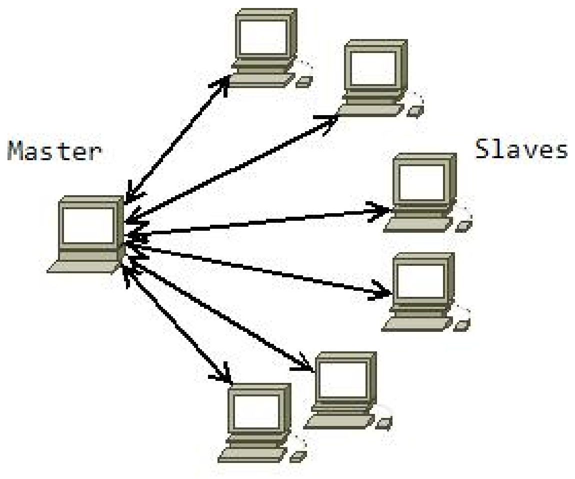

Figure 7: Communication relationship between master and slave cluster

The justification for this particular architecture is based on several factors. First, it would be advantageous for the system to run on consumer hardware rather than prohibitively expensive super-computers. Each node in the slave cluster was insignificant in terms of cost[19]. Second with respect to the parallel algorithm; while it was implemented as a distributed application each entity within the application[20] was inherently multi-threaded. For the master, the recipient and transmitter of many thousands of jobs, it was imperative that it performed in a timely manner with regards to the slave cluster; hence the choice of a multiple core processor – the ability to specify a listener and worker thread combination allows a softening of the communication overhead in that it may be presumed that the system has the capability of both receiving returned jobs and reallocating new ones concurrently. The slave application was also multi-threaded, with the logic for this being the same as that for the master; it is of benefit to the parallelization of the algorithm to be able to divorce working and listening. Another reason for these choices is the use of the 'shared-queue' (Mattson, Sanders & Massingill 2005, p. 183) pattern within the code itself; while other mechanisms were feasible, it seemed most logical to specify the relationship between threads in this fashion. It may also be obvious to the hardware savvy reader at this point that the slave nodes were single-core machines; this was a concession that was necessary when performing the

[19] Cost here is defined as financial cost. Note that this inference is relative; the cost of a slave node is inexpensive in comparison to the cost of a massively parallel supercomputer.
[20] Where *entity* refers to either a master or slave application within the overall architecture of the parallel algorithm implementation.

research. While the cost of multiple core consumer hardware continues to drop, it was not possible to assemble and test on such machines within the budget of the research. This undoubtedly had an effect on the measurable performance, as the slave application was deliberately designed to function better in an inherently parallel environment; the results section suggests the *actuality* of using these single core machines in the context of the parallel algorithm. Distributed algorithms also have the issue of network communications to address; the parallel algorithm demands a high number of communications that must succeed lest the resulting dynamic programming matrix be incomplete; as such, it is of great benefit to have a mechanism for transferring data with the knowledge that what is sent will arrive, and in the order that it was transmitted. TCP was chosen for this very reason; recall that TCP guarantees (ISI 1981) the delivery of packets in the order that they were transmitted, and that dropped packets are automatically retransmitted. It was considered worthwhile given the time available to sacrifice the speed of UDP for the reliability of TCP. The overhead introduced by TCP may have been a factor in the results obtained, however. Note that this does not remove the onus from the master and slave applications to ensure that the construction of the dynamic programming matrix proceeds according to the requirements of the parallel algorithm; that being, they must preserve the order in which blocks are computed. This particular issue is addressed by the previously mentioned use of the shared-queue pattern. The following section discusses some critical assumptions that were made when performing this research.

4.3.2 Assumptions

The measurement of the parallel exact algorithm's performance relies upon several assumptions about the behaviour of aspects of the distributed system as a whole. One of the primary assumptions is that by adding more nodes to the cluster, it may be possible to improve upon the run-time of the parallel algorithm. Whether this is the case or not is discussed in the results section, yet for the purposes of testing, an assumption was made that there was a point at which any speedup gained by parallelization of tasks would be outweighed by the cost of communicating between a large number of systems.

4.3.2.1 Networking

Another assumption is that the network bandwidth of a 10/100Mbit connection would be sufficient for the parallel algorithm's requirements. This derives from the concept that by using fixed block dimensions, it is possible to calculate in advance the maximum amount of data that is being transmitted at any one time. This calculation is a function of several variables. Thus, let b_j denote the bandwidth for a single job:

$$b_j = S(B)$$

Assuming that $S(B)$ is a function to give the size of a block in *bytes*. This presupposes that the contents of a block are known; this will be the case at all times for network communication:

$$S(B) = \mathit{dimensions}(B)\left\{L(C) \times \aleph = \sum_{f=0\ldots k+l} f\right\}$$

Where $\mathit{dimensions}(B)$ is the size of B in terms of block height and width; $L(C)$ is the length of all cells in block B, by which is meant the number of shapes in the set C; $\aleph$ denotes the sum of sequence identities, where there is an assumption that each sequence f in the composite alignment is numbered from 0 to $(k+l)-1$. In this case, k denotes the number of sequences in the first input alignment, and l the number of sequences in the second input alignment. This definition still requires certain blanks to be filled; for example, how are numbers stored by the implementation for different variables – the sequence identities are stored as unsigned shorts, given that it is impossible to have an identity that is below 0. This has the side effect of preventing the parallel algorithm from being able to compute alignments where there are greater than 65536 sequences. It is assumed that a `ushort` will require at least 2 bytes for storage. Being aware of these factors is one step to knowing required network bandwidth; however an additional factor that may introduce uncertainty into the calculation is that of serialization.

4.3.2.2 Object Serialization

The mechanism employed for transporting logical objects across the network was bit stream serialization. User-identifiable objects are translated into a memory stream that is more suitable for transmission to a remote host. The process of serialization requires that metadata of the serialized object be maintained along with the object itself (Microsoft 2007a) – this includes but is not limited to elements such as the size in bytes of the object, and meta-data about the contents of its fields. Simply put, the act of serializing an object can cause 'bloat' that may be unwanted, but is required in this circumstance; thus it is difficult to accurately predict how large a job will be after it is serialized.

4.3.3 CALCULATION OF RESULTS

This section describes the process by which results were calculated. It assumes that the reader is familiar with the background architecture of the parallel algorithm as presented earlier in this section.

4.3.3.1 Preprocessing of data with ClustalW

The input to the exact algorithm is defined as being two well-formed multiple-sequence alignments. The samples used for testing were obtained from the BAliBASE 3.0 (Thompson J. D. et al. 2005) first reference set. This dataset contains a mixture of potential alignments; some that are extremely small while others are quite large. By incorporating a variety of alignment sizes in the tests, it becomes possible to examine the extent to which the parallel exact algorithm achieved a speedup over the sequential algorithm and where it did not.

Before the BAliBASE test data could be used, however, it was necessary to generate the multiple-sequence alignments that would be used for testing from the raw sequences contained in the dataset. This was achieved by pre-processing the input data with ClustalW (see section 2.3.2.2 for a description of this program). ClustalW takes a number of parameters that specify how to perform the alignment; for the purposes of testing, all values were left at their default except for the following fields: fast pair-wise alignments were enabled (resulting in a loss of some accuracy in terms of mathematical correctness), and fasta file output was specified. In all cases, the Gonnet (1992) protein weight matrix series was used when pre-processing the input data. It should be noted at this point that no emphasis was placed on proving the quality of alignments produced by the parallel exact algorithm; as it will produce identical alignments to the serial exact algorithm as reported by Kececioglu and Starrett (2004). While accuracy was not a concern, composite alignments *were* generated in the interests of ensuring that the algorithm implementations were correct. These can be found in the appendices, along with the initial input alignments generated by ClustalW.

An additional point to make about input data is that in order to evaluate whether the algorithm has performed as expected, the output must be compared to existing known alignments. Therefore, the tests run upon the parallel exact algorithm are designed to split an existing alignment and *polish* it by realigning, thereby allowing the final output to be compared against the BAliBASE reference alignments (as contrasted with the BAliBASE unaligned sequences). When splitting an output alignment from ClustalW, it was assumed that the number of sequences in an alignment would be even (thus, a 50/50 split would be performed). If this were not the case, alignment A would be assigned one more sequence than alignment B, a factor that is unlikely to affect the performance of the algorithm in that it is the number of identities (or characters) in a sequence from an alignment rather than the number of sequences themselves which dictates the size of the dynamic programming table that is constructed. The following sections discuss how testing was performed.

4.3.3.2 Serial AAE

As the objective of this research is to examine how useful any parallelization of the exact algorithm might be, the first set of tests run is upon the sequential algorithm. Time samples were obtained by running the RV1 dataset through the sequential algorithm and calculating the time spent composing the dynamic programming matrix. The mechanism for obtaining time elapsed is discussed in section 4.2. As previously mentioned, several measurements were taken, being the time in fractional minutes and seconds, and sizes of input alignments used to create the programming matrix. The mean execution time of the serial algorithm was recorded after 5 runs over each alignment in the RV1 dataset. Given that this dataset contains highly divergent sequences[21], a BLOSUM45 (Henikoff & Henikoff 1992) cost matrix was employed. A copy of the matrix employed is included within the appendices.

4.3.3.3 Parallel AAE

The evaluation of the parallel algorithm was conducted over a variety of system configurations. Initially, it was tested with a single slave node; this was to establish the performance differential between the serial algorithm and the parallel. As with the serial algorithm, tests were performed 5 times on each alignment, with the mean time being recorded for comparison. Following this, the algorithm was tested over 2, 3, 4, 5 and 6 slave nodes, with the intention that any speedup could be calculated by comparing the serial algorithm performance on a given alignment with the parallel performance for a predefined number of slaves. As with the serial algorithm, a BLOSUM45 cost matrix was employed to account for highly divergent sequences. The parallel algorithm also requires additional parameters to compute an alignment; namely the height and width of a block respectively. The height of a block is also the height of a stripe in *cells*. For all tests performed on the exact algorithm, a fixed block size of 20×20 cells was employed: a number chosen for two reasons. First, refer to section 3.2.1.3 for the justification for attempting to secure a relatively high computation to communication ratio. Second, the performance of the parallel algorithm over small alignments was predicted to be inferior to that of the sequential algorithm; as such, the chosen block size is targeted towards improving the performance of the parallel algorithm over medium to large alignments. The definitions for 'small', 'medium' and 'large' alignments are contained within the results section 5.1.1. The parallelization of the exact algorithm is therefore, presumed to not improve the computation time over small alignments *in principle* based on both the size of those alignments and the predefined block dimensions specified here. Actual performance given this block size is recorded in section 5.1. Section 5.2 speculates on alternative results that may have

[21] Highly divergent refers to the level of conservation in the sequences. See section **3.3.4**

been garnered given two different approaches: first, a non-fixed block size that is calculated dependent on the size of the input alignments and the number of pUnits available; second, an examination of the performance of varying block sizes over the different alignment classes (small, medium, and large).

4.3.4 SUMMARY

This section discussed the approach used to evaluate the performance of the parallel exact algorithm. The following section showcases the results obtained using this method, and presents a discussion on their characteristics and speculation about the utility of the approach described above.

5. RESULTS AND OBSERVATIONS

This section contains the results of experimentation on the parallel exact algorithm and a comparison of its performance against the serial exact algorithm. Some results obtained diverged from expected outcomes as discussed in the section on logic (3.2); in these cases, observations are made as to why certain of these results appear flawed and what this means in terms of the parallel algorithm's overall performance.

5.1 PERFORMANCE COMPARISON

In order to compare the performance of the parallel exact algorithm to that of the sequential algorithm, it is necessary to calculate how much overhead the parallel algorithm introduces. Section 4.3.2 discusses critical assumptions made about the implementation, in particular that it was assumed that the bandwidth of a 10/100Mbit network connection would not serve as a bottleneck to the algorithm's performance. In terms of performance, however, the stated primary metric is that of time. Therefore, in order to determine the overhead of the parallel algorithm versus the performance of the sequential algorithm, it is necessary to take the mean performance of the parallel algorithm over a single node and contrast this against the mean performance of the sequential algorithm on the same data instances.

5.1.1 PARALLELIZATION OVERHEAD

The RV1 dataset introduced in section 5.3.3.2 can be divided into two sub-sets for the purpose of determining the relationship between sequential and parallel (1 node) algorithm performance: RV11 and RV12, where RV11 contains alignments composed of sequences from distantly-related organisms and thus little homology; and RV12 where alignments display between 20-40% homology (Thompson J. D. et al. 2005). Table 1 displays the mean processing time of both the sequential and parallel (with a single node) algorithms over the dataset RV11. It is perhaps unsurprising that the parallel algorithm with a single node underperforms the sequential algorithm by approximately 25%. This is likely to be an indication of the inherent overhead involved in parallelization and distribution of the matrix composition process.

	Processing Time $(mean\ secs)$	Number of Cells $(mean)$
Sequential	52.99643	172644.3
Parallel (1)	70.23393	
Speedup factor	0.75457	

Table 1: Speedup factor as a function of mean processing time over RV11

The underperformance of the parallel (1 node) algorithm over the RV11 sub-dataset may not be representative of its performance in all circumstances, especially given the highly-gapped and low-homology alignments that RV11 represents. Table 2 demonstrates the relative performance of the algorithms over RV12. This sub-dataset is composed of a number of larger instances than RV11, resulting in a higher mean cell count for output programming matrices. Given that there is a higher degree of conservation in the alignment instances from the second dataset, this may have affected the resulting processing times.

	Processing Time $(mean\ secs)$	Number of Cells $(mean)$
Sequential	140.7969	362125.2
Parallel (1)	152.3723	
Speedup factor	0.9240	

Table 2: Speedup factor as a function of mean processing time over RV12

Somewhat unexpectedly, the mean processing time for the parallel (1 node) algorithm over RV12 appears to be approximately only 8% worse than that of the sequential algorithm. This is a more positive result than the one achieved over $\mathrm{RV11}$, yet is equally untrustworthy; the variation in both size and complexity of alignments in the two data sets meaning that conclusions ought not to be drawn from these two results. Taking the mean of these two values results in a speedup factor of 0.8393, a value that is more likely to be representative of the general performance of the parallel algorithm over 1 node.

It should be mentioned, however, that further experimentation would be instructive in this matter, perhaps over the remainder of the BAliBASE. Given an expectation that in general speedup over the sequential algorithm is going to have to offset by an approximate 20% loss of efficiency due to communication and marshalling overheads, the following sections examine the performance of the parallel exact algorithm over a variety of alignments. These results are divided into three categories, small alignments – where small denotes a table of $< 10^5$ cells, medium alignments – where medium denotes a table of $\geq 10^5$ and $< 10^6$ cells, and large alignments – where large denotes a table of $\geq 10^6$ cells. It is important to recognize that this categorization of alignments is superficial; determining the threshold at which the parallel algorithm becomes worthwhile would be the objective of defining a range; however it would be irresponsible to claim knowledge of this threshold given the limited range of test data employed. A more comprehensive test over the whole BAliBASE would provide a more trustworthy representation of its performance. The following sections present some initial conclusions given the available data.

5.1.2 Small alignments

Parallel versus sequential time performance is a simple comparison to make, however it is probable that benefits are not applicable to all alignments. For small alignments, the existing sequential computation time is already somewhat trivial. Section 3.2.1.4 describes limiting factors in any potential speedup for the parallel algorithm; one being the number of pUnits used for a particular alignment versus the number of stripes in the set of stripes Z. Section 5.1.1 defined the size range of a small alignment to compose a table of $< 10^5$ cells. Displayed below are the results of the parallel exact algorithm's performance versus the sequential algorithm over 3 alignments from the 'small' range. These samples are selected from $\mathrm{RV11}$ and $\mathrm{RV12}$, and provide a suggestion of the performance of the parallel algorithm on alignments of a minor size. The instances selected represent the lower, mid-range and upper bounds of the 'small' alignment range as defined in 5.1.1.

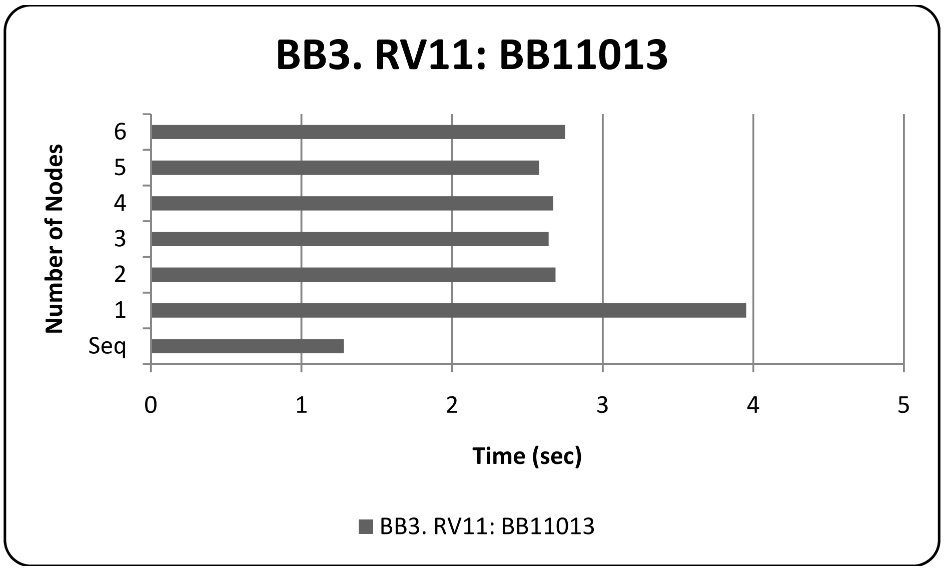

Figure 8: 13th instance of RV11, performance over pUnit scaleup

The performance over the smallest instance in the RV11 dataset is summarised by Figure 8. A visibly poor result of the single node parallel algorithm was softened by the eventual speedup obtained by adding more nodes to the cluster. Interestingly, this chart shows a continuing trend over the results; two systems provides a speedup, but adding more systems does not noticeably improve the performance of the parallel algorithm overall. Figure 9 shows a similar trend; the performance of the sequential algorithm is noticeable better in these smaller instances, despite the size of instance 28 being nearly a factor of 6 larger than instance 13 in terms of raw cell count.

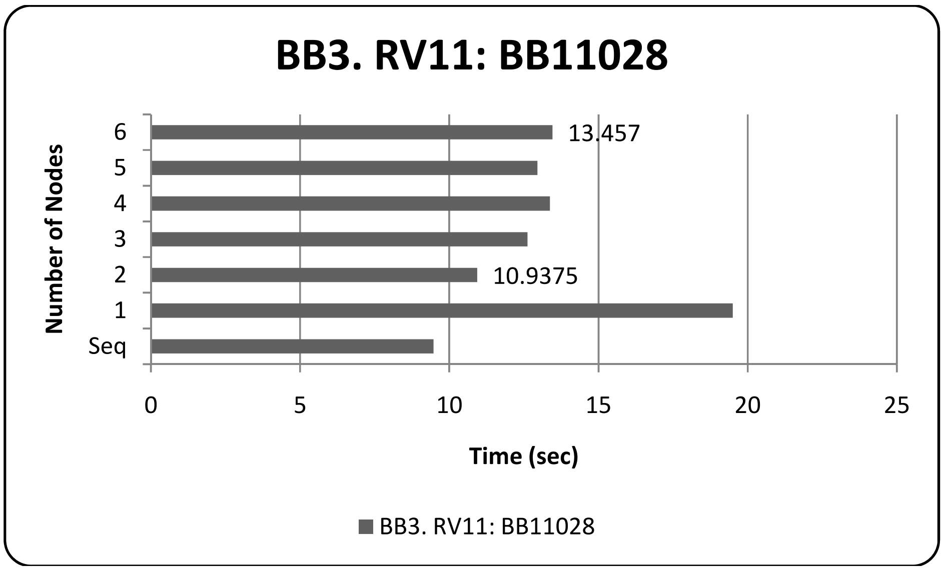

Figure 9: 28th instance of RV11 performance over pUnit scaleup

Also interesting about Figure 9 is the performance of two pUnits versus the scale up over more. This appears to support the supposition made in section 3.2.1.4 that the addition of pUnits to the cluster may in some cases *worsen* the performance of the parallel algorithm, though this instance by itself is not enough to confirm this suspicion. Figure 10 describes behaviour over the largest of instances in the small alignment interval, having some 8.289×10^4 cells.

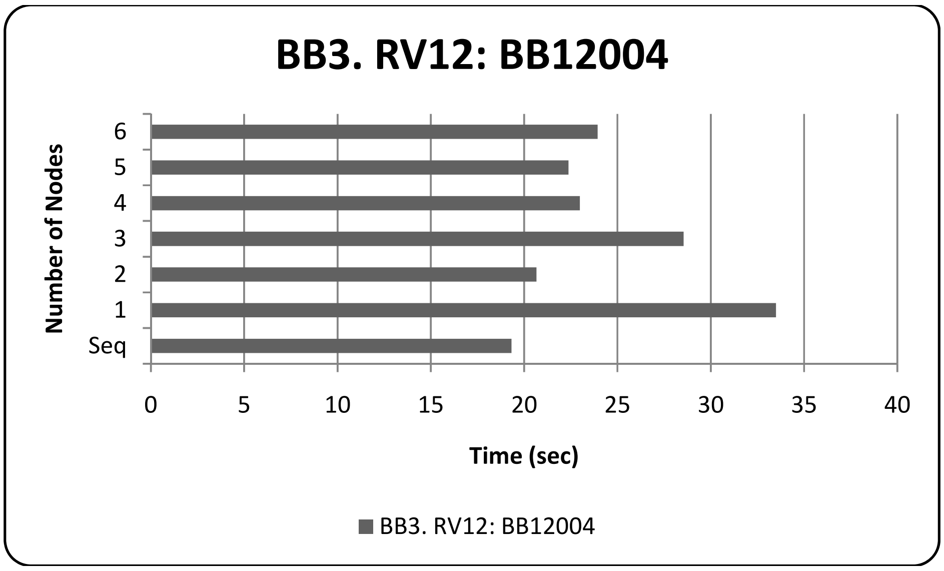

Figure 10: 4th instance of RV12, performance over pUnit scaleup

As with the previous samples, the performance for the 4th instance of RV12 worsens with the addition of more pUnits. Curiously, when run over three pUnits, the performance was noticeable worse than when it was run over four pUnits. This is likely to be an artefact of a limited sample size (5), and thus unlikely to represent a more general trend. In light of these results, the initial performance of the parallel exact algorithm appears disheartening; recall however, that the objective was to investigate how worthwhile any parallelization might be. While it is not possible to make a generalisation about the performance of the algorithm on small instances given the size of the dataset employed, it does seem apparent that performance over small instances in RV1 is not very promising. It is important to recall, however, that these instances from the 'small' range are relatively trivial in size compared to other instances in the dataset. Performance over medium and large alignments is discussed in the following sections.

5.1.3 MEDIUM ALIGNMENTS

A medium alignment in the context of the RV1 dataset was defined in section 5.1.1 as contributing to a dynamic programming table larger than 10^5 cells and smaller than 10^6 cells. The medium class represents a much larger number of alignments than does the 'small' class; this is in an attempt to categorise the most commonly occurring instances in the RV1 dataset. These results are particularly interesting, in that they show a significant speed improvement for the parallel exact algorithm over the sample instances in the upper two thirds of the 'medium' range. Figure 11 describes the performance of the 15th instance from the RV11 dataset. This particular instance was selected to represent the general performance of the parallel algorithm over alignments falling into the first third of the 'medium' range. As shown in the figure, sequential and parallel (2 node) run times have been labelled to illustrate the relatively minor difference between the two. Figure 10 showed a similar trend, which seems logical given that the two instances are only separated by some 2×10^3 cells. At this point, it also appears that the additional overhead of spreading the computation task over more than two nodes results in non-trivial time penalty of approximately 5 seconds. Perplexingly, this behaviour appears relatively constant when scaling from 3 to 6 nodes. It should be noted that it is not necessarily the case that timeliness is not improved in these circumstances: it may be that any speedup that would be possible by scaling up to n nodes is being countered by the additional communication costs involved in the process of scaling up.

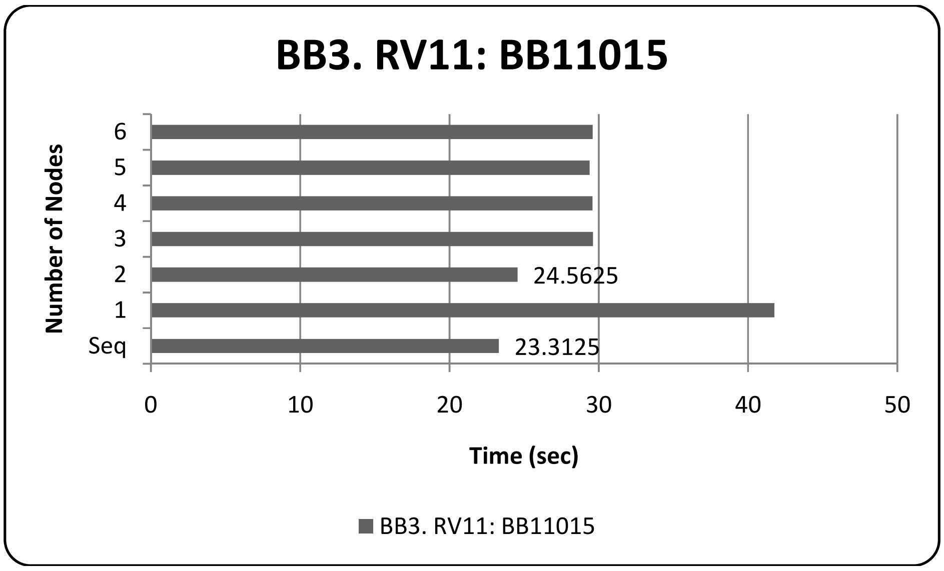

Figure 11: 15th instance of RV11, performance over pUnit scaleup

The second sample represents a mid-sized alignment from the medium class. At 4.4442×10^5, the matrix composed from the 43rd instance of RV12 is nearly a factor of 5 larger than the alignment represented in Figure 11. This is a result of the much larger interval of the class of 'medium' alignments. Figure 12 demonstrates the performance of the parallel algorithm over this particular instance. Surprisingly, it is considerably better in terms of relative processing time than the previous sample might have suggested. The speedup factor on the parallel algorithm (6 node) over the 43rd instance from RV12 is 3.35 with relation to the sequential performance. This is in comparison to the speedup factor on the 15th instance from RV11 which was a meagre 0.79. Some interesting conclusions could be drawn at this point; it would be premature, however, without examining the instances from both the 'large' class and indeed the final sample instance in the 'medium'.

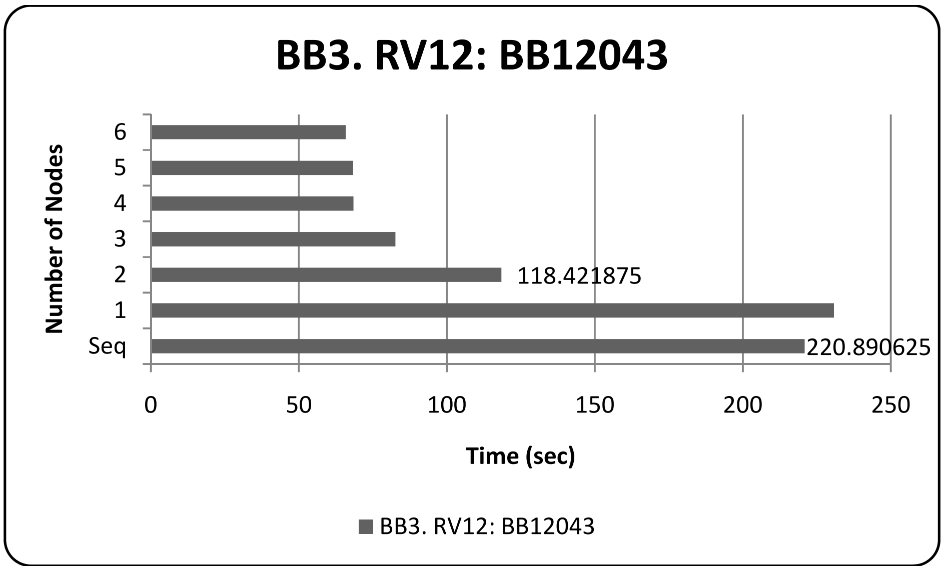

Figure 12: 43rd instance of RV12, performance over pUnit scaleup

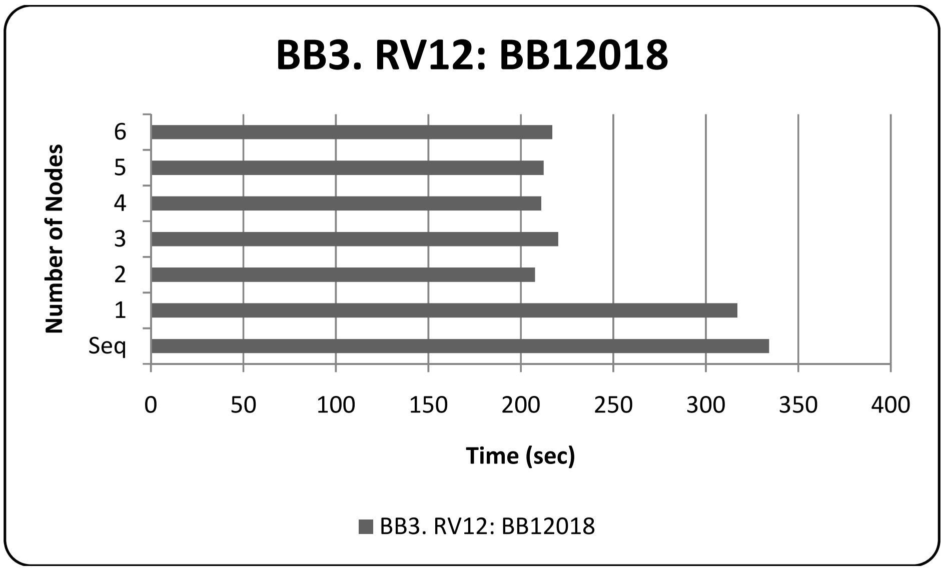

Figure 13:18th instance of RV12, performance over pUnit scaleup

The final sample for the 'medium' class is the 18th instance from the RV12 sub-dataset. This instance is once more substantially larger than its predecessor; composing to a table of 7.80263×10^5 cells. As with the previous sample, there is a visible improvement in the performance of the parallel algorithm versus that of the sequential. Conversely, adding more than 2 nodes in this circumstance does not appear to have improved performance at all. Given two nodes, the performance improves by a factor of 1.61 in comparison to the same instance processed sequentially; yet again this is not on the scale of that of the previous sample – a factor of 2 worse than the maximum speed up possible on the 43rd instance. Another peculiarity is the performance of the parallel algorithm with a single node; it appears to have achieved a better result than the sequential algorithm. The varied performance of the parallel algorithm over 'medium' class instances is cause for speculation. Section 5.2 discusses this in more depth. Given the performance on the largest instance in this category, it would be instructive to see how well the parallel algorithm performs over large instances. This is the topic of the next section.

5.1.4 LARGE ALIGNMENTS

Recall that section 5.1.1 defined a 'large' alignment as being one that composes to a dynamic programming table with greater than 10^6 cells. This class represents the upper size bound of instances from the RV1 dataset and was expected to take significantly longer to process than its smaller counterparts 'small' and 'medium' respectively. It should be noted that in terms of representation, the 'large' alignments sampled here are all from RV12; this is due to the largest alignment in RV11 being sub-10^6 cells. The first sample is the 38th instance of RV12, with 1.056048×10^6 cells. Figure 14 represents this particular alignment's results.

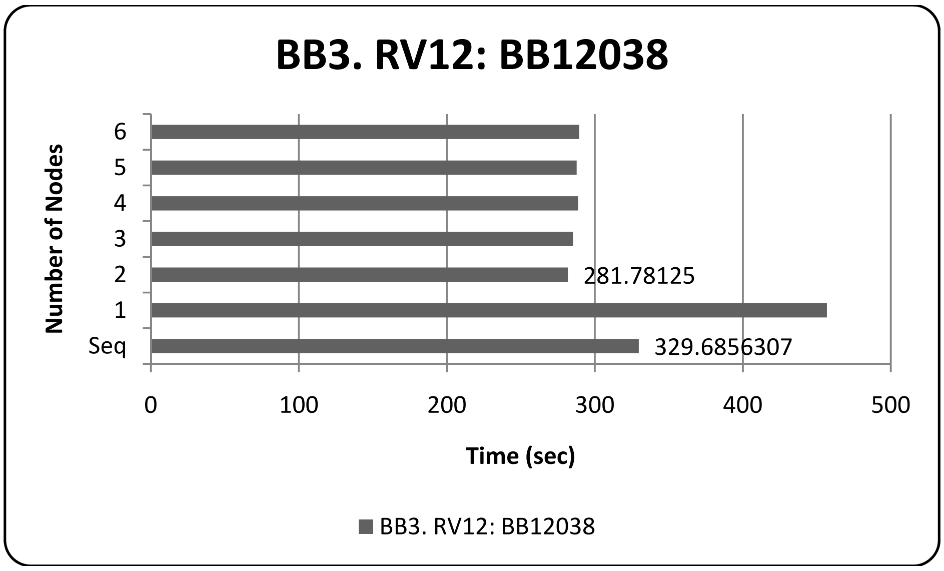

Figure 14: 38th instance of RV12, performance over pUnit scale up

The observable performance improvement for this instance is somewhat less apparent than that displayed in the 43rd instance (section 5.1.3). The best parallel result was obtained over two nodes and is approximately 14% faster than the sequential algorithm's result. This relates to a speedup factor of 1.17. Unlike the final sample in section 5.1.3, the performance of the parallel (one node) algorithm in the worst case is poorer than that of the sequential algorithm. This appears more in line with earlier results over smaller instances. The second case for the 'large' range is the 7th alignment from RV12. This alignment composed to a matrix of 1.3221×10^6 cells. Performance is presented in Figure 15.

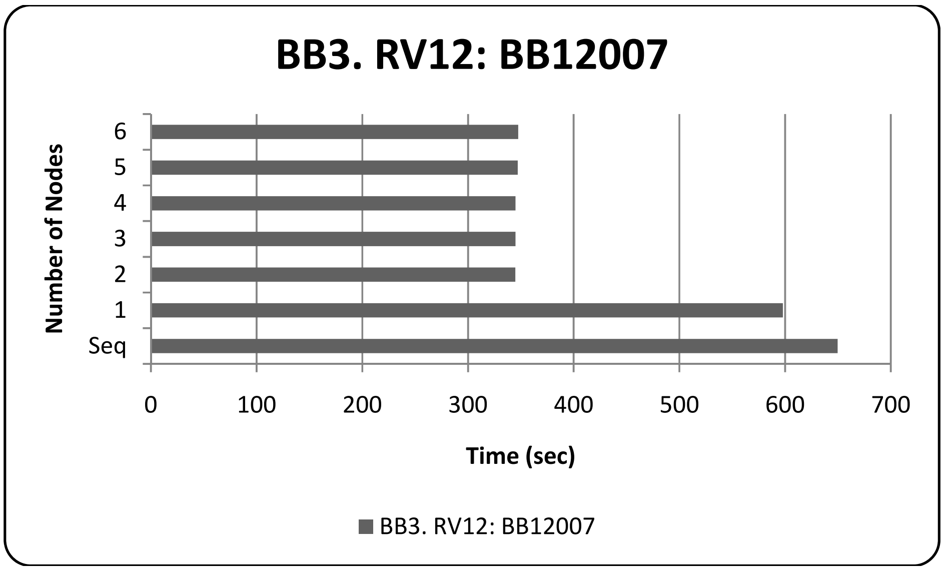

Figure 15: 7th instance of RV12, performance over pUnit scale up

Interestingly, this sample displayed the same phenomenon as that observed in the final sample of section 5.1.3; namely, an *improvement* in timeliness over a single node running the parallel algorithm. Perhaps more interesting is the extent to which performance was improved by adding an additional node to the cluster (hence, two nodes in total): a speed up factor of 1.88. In percentile terms, this equates to an improvement over the performance of the sequential algorithm of approximately 46%. This is slightly over a factor of 3 better than the performance improvement in the parallel results of the initial sample for the 'large' range. The final sample for this range is the 37th instance of RV12 and perhaps more importantly, the largest of all instances tested. The table composed in this case was 2.462528×10^6, over 10^6 cells larger than the next largest in the data set. Performance over this alignment is summarized in Figure 16.

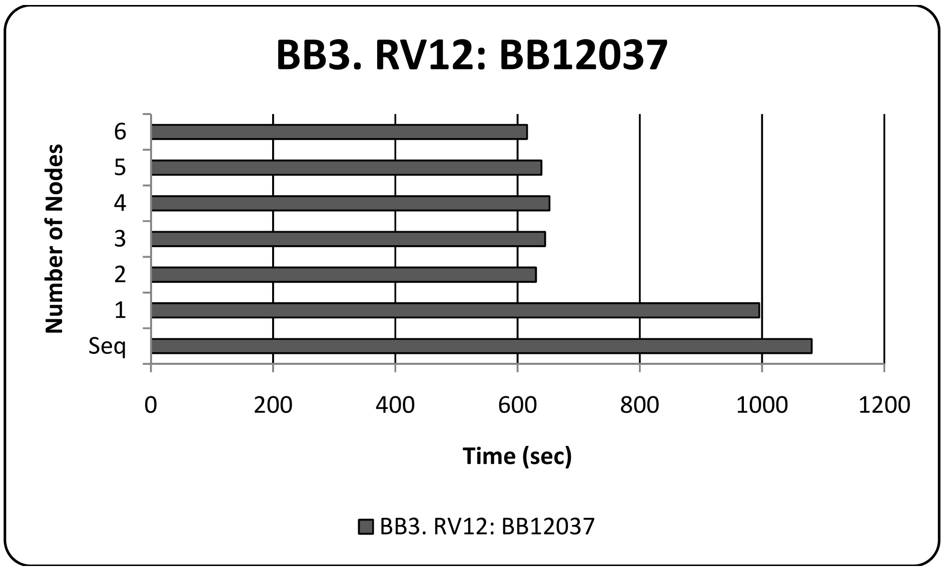

Figure 16: 7th instance of RV12, performance over pUnit scale up

As with the second sample for this range, timeliness is improved even after initially applying the parallel algorithm with a single node. Unlike previous examples where it was observed that having two nodes was generally better than scaling up, in this circumstance 6 nodes saved approximately 15 seconds of processing time. Despite this, the best case time for computing the table over this alignment was still greater than 10 minutes. To consider the result more optimistically, 10 minutes is still markedly better than the sequential processing time, which clocked on average 18 minutes. In terms of speed up, the 8 minute difference in time equates to a factor of 1.76 -- slightly worse than the parallel algorithm's performance over the previous sample with only 2 nodes. It seems clear that there are benefits to be had from parallelizing the composition of the dynamic programming matrix, but at this point it is difficult to make a generalisation that is likely to hold in all cases. Section 5.2 goes into more depth on this issue. At this point, given the examination of individual alignment performance, it is instructive to consider the performance of the parallel algorithm over a wider range of instances.

5.1.5 Speed up in general

In order to make conclusions about the performance of the parallel algorithm in general, it is necessary to examine its performance over many tasks. The tables in this section summarise the results of the parallel algorithm versus those of the sequential for each of the ranges described in section 5.1.1. To aid analysis, a mean performance improvement factor is provided for each class. Table 3 represents the 'small' class of alignments, with a mean cell count of 39261.33. This particular range is composed of some 30 instances. As shown in the figure, the best recorded performance by the parallel algorithm was over two nodes; however, this was still sub-optimal given that it underperformed the serial algorithm over the same range by some 2 seconds. For the small instances of RV1 it would appear that parallelization yields no overall benefits. With further data, it might be possible to generalise this trend to all small alignments.

'Small'	Mean Time (sec)	Speed up	Mean Cells
Sequential	7.842708	N/A	39261.33
Parallel(1)	17.24479	0.454787	
Parallel(2)	9.967188	**0.786853**	
Parallel(3)	12.22344	0.641612	
Parallel(4)	12.33906	0.6356	
Parallel(5)	11.4651	0.68405	
Parallel(6)	11.4418	0.685444	

Table 3: Mean run time and speed up over RV1 dataset ('Small' range)

'Medium'	Mean Time (sec)	Speed up	Mean Cells
Sequential	95.65859	N/A	288758.7
Parallel(1)	119.3934	0.801205	
Parallel(2)	74.27188	**1.287952**	
Parallel(3)	78.73672	1.214917	
Parallel(4)	78.59219	1.217151	
Parallel(5)	78.73086	1.215008	
Parallel(6)	77.67422	1.231536	

Table 4: Mean run time and speed up over RV1 dataset ('Medium' range)

'Large'	Mean Time (sec)	Speed up	Mean Cells
Sequential	601.695	N/A	1360250
Parallel(1)	568.7292	1.057964	
Parallel(2)	355.8229	**1.690996**	
Parallel(3)	365.4737	1.646343	
Parallel(4)	356.8012	1.686359	
Parallel(5)	355.875	1.690748	
Parallel(6)	357.1756	1.684591	

Table 5: Mean run time and speed up over RV1 dataset ('Large' range)

Table 4 provides a summary of parallel performance over slightly larger alignments from the 'medium' class. Unlike the 'small' range, the use of the parallel algorithm over these alignments yielded a noticeable improvement in timeliness. This trend was suggested by the samples examined in section 5.1.3 though it is clear from the speed up value over all 40 instances in the 'medium' range that the improvement was marginal. This is represented by a time saving of approximately 22% versus the sequential algorithm in the best case (2 nodes). Table 5 exemplifies the performance over the 'large' class. The parallel algorithm's performance in this instance was slightly more optimistic, with the best case improvement being 1.69 times faster than the performance of the sequential algorithm. It should be noted that the sample size of the 'large' class was disproportionate to the 'small' and 'medium' classes, given that it contained only 6 alignments. This biases the result somewhat, meaning that it would not be wise to generalise on the performance of the parallel algorithm over this particular range of instances. The following section provides speculation as to the validity of the results discussed and some suggestions for how the parallel algorithm might be improved to show a performance increase given a higher number of pUnits.

5.2 Discussion

Section 5.1 presented a summary of the results obtained during the course of this research. Certain aspects of those results warrant further investigation while other aspects deserve speculation given that they were not expected. This section attempts to reconcile the results presented and explain some of the apparent irregularities in the sampled instances.

5.2.1 Parallel algorithm performance

Section 5.1.5 describes the performance of the parallel algorithm over the three defined classes of 'small', 'medium' and 'large' alignments. It was shown that for alignments that compose to a table of less than 10^5 cells, that for the $\mathrm{RV1}$ dataset, performance is somewhat retarded by the application of the parallel algorithm. This is likely to be due to factors discussed in 3.2.1.4, such as the overhead of communicating the job to be processed between many nodes. Given that the best case performance of the parallel algorithm was over only two nodes, it is apparent that the further overhead incurred through additional parallelization (the adding of more slave nodes) more than offset the benefits that may have been gained through such an action. Another variable that may affect the performance of exact algorithm in general is that of shape-list length. Recall in section 3.1.2.2 the definition of dominance pruning. In the best case, dominance pruning results in shape-lists of length 1. This is not a claim of the average or worst case number of shapes generated; the number of which may be considerably larger than 1. Consequently, when examining the results from section 5.1.3, it was observed that one instance performed notably better when scaled over many machines than did another of similar matrix proportions. It is not improbable that the instance providing a greater speedup factor displays shape-list characteristics that render it particularly well-suited for this kind of parallelization. Determining *favourable* matrix characteristics would be instructive in improving the performance of the system in general.

A critical factor that has not been examined is the effect that the number of columns in the programming matrix has upon composition efficiency. Logically, the number of columns in table T is proportional to the number of blocks in some stripe ζ. The reduction in the number of blocks in this fashion results in less time being spent on each stripe. Intuitively, it would seem that this is a *good* thing, and that this would result in a more rapidly composed table. Conversely, the transition from stripe to stripe involves additional communication between master and slave. Interpretation of the parallel algorithm's performance when scaled over more than 2 nodes suggests that this overhead is significant enough to prevent a gradual speed up from being noticeable on small instances. This behaviour is apparent in row striping also; where there are few rows in a table, fewer stripes are defined. The presence of fewer stripes dictates that it may not be possible to use all pUnits available at any one moment. Thus, for any given alignments A and B, if the number of characters m from a sequence in alignment A is a relatively minor number, so too will the number of stripes be limited. Unfortunately, there is nothing preventing a table from being wholly asymmetrical: there could be half as many rows as columns. In this circumstance, a pUnit would spend more time on a single stripe. It might not be possible to scale up the parallelization because of limited stripes. In an extreme (and improbable) case, imagine a table with 21 rows and 2000 columns. The parallel algorithm requires a single row to pre-compute input values for the subsequent blocks in stripe 0; this results in only a single stripe being generated (if we assume a 20×20 block size as was used during experimentation) that contains 100 blocks. Each block composes to 400 cells, which equates to a single node computing 4000 cells even if there existed > 1 node when the alignment was initiated.

Despite this limitation, over 'medium' and 'large' alignments the parallel algorithm's performance was more acceptable (by acceptable, less costly than that of the sequential algorithm) in general which raises the question of how and why is this possible? The answers to these particular questions are inherent in the logic defined in section 3.2. One aspect of the answers is the inverse of the situation suggested earlier where there are a minor number of columns; essentially, the longer a pUnit spends on a particular stripe, the more beneficial parallelization appears to be. How does this follow? Recall that the act of moving a pUnit to a new stripe is not a cheap activity – it requires the Master of the distributed system to check who is processing what and make an intelligent decision based on this as to whether or not to assign a new stripe. Concurrently, processed jobs are returning to the master, which must then process these and produce *new* jobs, if necessary, to be redistributed. It follows that the more time the Master spends housekeeping, the less time it has to allocate and receive jobs from its slaves respectively.

Therein lies the critical problem with the adopted approach to parallelization; that a single component of the distributed system (the Master controller) may unwittingly function as a bottle-neck that impedes potential processing speed improvements. The Master is not the only culprit, however; the slave nodes assume at all times that they are processing a job that continues the propagation of a particular *stripe*. If it is the case that they receive a job that is *not* a continuation but rather the beginning of a new stripe, minor adjustments to their internal structure must be performed that again, result in a slight time penalty. This speculation, however, requires further investigation to determine actual bottle neck locality. The following section discusses the sample instances that were examined and why certain of these displayed irregular characteristics.

5.2.2 Thresholds and Unpredictability

When analyzing the results for the parallel algorithm versus the sequential algorithm, the first instance that displayed better performance on the part of the parallel algorithm was the instance discussed in section 5.1.3, the 43rd instance of RV12. Recall that this sample was 5 factors larger than its predecessor. This was due to the larger interval of the 'medium' class than that of the 'small' and 'large' classes respectively. The sample examined before this was the 15th instance of the RV11 dataset. The results for the 15th instance suggested that at some point the performance of the parallel algorithm might surpass that of the sequential – given a large enough table; a supposition that relied on the belief that there was a change in trend over time from the initial samples examined in the 'small' range to those in the 'medium' range. The 43rd manifested a dramatic swing in performance, with the parallel algorithm achieving a large speed improvement versus the sequential. What remains unclear, however, is the *threshold* at which the parallel algorithm becomes more efficient than the sequential. Presumably this threshold can be induced to exist for table sizes somewhere between the size of the 15th instance of RV11 and the 43rd instance of RV12. Further experimentation would be required to determine the location of this threshold. The dataset employed provides suggestions to its whereabouts, yet does not contain enough instances to accurately map the final location. Perhaps more confounding are the number of variables that may contribute to a boost in the parallel algorithm's performance. As mentioned in the previous section, further profiling of the algorithm and associated distributed system is required to determine its true bottlenecks; this would be a pre-requisite before attempting to establish the existence of a general threshold.

5.2.2.1 Unexpected outcomes

Certain results from the 'large' class displayed a trend that was somewhat perplexing. Recall that in section 5.1.1, a general rule was established that the performance of the parallel algorithm using a single slave node was inferior to that of the sequential algorithm. This was due to the additional overhead required for the parallel algorithm to complete a task. Surprisingly, the two final samples chosen for the 'large' range display a time for the parallel algorithm (one node) that is somewhat better than that recorded by the sequential algorithm. Initial thoughts were that the test regime of performing each alignment 5 times was insufficient. A 10-fold or even 100-fold test would have been highly preferable, but were impossible due to time constraints. Therefore, it is not possible to rule out an aberration in the mean time elapsed calculation – yet this would still require the parallel (one node) algorithm to surpass the performance of the sequential algorithm at some point.

Alternatively, the sequential algorithm might have performed especially poorly on those alignments; this too could be a result of a small sample size. More interesting perhaps, is the possibility that it did indeed compose the matrix in a briefer period than did the sequential algorithm over the same alignment. Logically, this should not be possible – yet granted the non-trivial length of the columns in the 'large' samples, it may be the case that the memory consumption for the sequential algorithm (over 300mb when composing the largest of alignments, instance 37 from RV12) resulted in the slowdown of the algorithm for these extra large instances (where the number of cells C is 10^6 or greater). This is plausible assuming that the parallel slave nodes never store more cells than the block that they are computing at any given time. Because of this fact, over the tests performed each client at any one time would not need to store more than 400 cells worth of data, a relatively trivial amount. The master node must store the eventual output from the slaves; however it does not perform any operations upon the data, simply writing it to a final matrix which is then untouched until the final traceback procedure. For the purposes of testing, the traceback was untimed as it was only possible to improve performance on the composition portion of the algorithm. Given this speculation, the following section describes an alternative method for calculating block sizes and hypothesizes on the potential effectiveness of this.

5.2.2.2 Variable block sizes

One element of the parallel algorithm that warrants further investigation is that of block size variability. It is improbable that a 20×20 block is the most efficient size for all alignments; on the contrary, this size was chosen because it provided understandable and comparable results. If it were the case that the parallel algorithm was to be used in a real laboratory environment, it would be beneficial to investigate potential speed up based on block size. It is possible to calculate the most appropriate block size based on several factors; size of respective input alignments A and B, number of pUnits available for use as slaves, distribution of rows and columns (are there more rows, are there more columns) and finally, maximum effective network bandwidth given n slave nodes. It is probable that other factors would play a role in determining the most appropriate block size, but these will remain enigmatic without further research. Unfortunately, this logic only extends so far. The existence of a minimum useful block size has not been determined through the research conducted for this document; however the expression in section 2.2.2.2 describing the communication to computation ratio dictates the existence of this minimum size with respect to potential speed up.

5.2.3 Summary

This section described the results obtained throughout the course of research. Conjecture was provided where results did not appear to follow the general trend. It was suggested that the maximum speed up factor for the parallel algorithm may be constrained by unidentified bottlenecks. Further research is required to accurately identify these bottlenecks and improve the performance of the parallel algorithm in general. The following section concludes on the work performed.

6. Conclusions

This research investigated the extent to which the AAE algorithm could be effectively parallelized, as stated in the hypothesis. Initially, consideration was given to previous work in the field. The literature review examined the field of molecular sequence alignment, with an emphasis on the exact algorithm itself. Also discussed were issues inherent when parallelizing an algorithm; speedup factors and communication costs that were instrumental in allowing this research to be conducted.

The adopted implementation approach was one of distributed parallelism, with an emphasis on minimising application cost while attempting to achieve a pragmatic improvement in run time over non-trivial alignments. In order to collect results, tests were performed over a small cluster of 6 machines with a master node arbitrating activities between slaves. Each alignment was processed 5 times with the mean time for this processing stored for analysis. For comparability, the sequential algorithm was re-implemented in the same language (C#) as the parallel algorithm's implementation, and tested upon a single node that was later employed as a slave in the cluster.

Initial results were disheartening, with the parallel algorithm failing to deliver a speedup over minor alignments. This was not an overall trend, however, as the algorithm showed that for medium to large alignments (as defined in section 5.1.1), there was a noticeable speedup when parallel performance was compared to that of the sequential algorithm. In the best case, the parallel performance was a factor of 3 better than that of the sequential. The initial hypothesis is satisfied to some degree, with the performance of the parallel algorithm implying that in certain cases it is beneficial to break the composition of the dynamic programming matrix up and process it in parallel. Unfortunately, it was also the case that due to a lack of data, it was not possible to generalise this trend over a wider range of alignments other than those which meet strict size criteria. Furthermore, it was not shown that scaling beyond more than 2 slave nodes and a single master provided any improvement in performance. Despite this, a close to optimal speed up for the 2 slave case was a promising result in some respects. The following section discusses further work that could be undertaken with respect to this research.

7. Further work

Given the results obtained from this research, further work may be divided into two categories. The first of these would attempt to improve the performance of the parallel algorithm in the environment that it was initially tested in; a distributed multi-system master-slave arrangement. This has the advantage of being tried and tested; it also has several questions already posed that can be answered with a little more research. There is at least one disadvantage to proceeding down this path; that of maximum scale up. It is probable that the maximum speedup achievable using this approach is significantly poorer than the approach described in the following paragraph.

The second aspect of further work would be a complete re-envisioning of the algorithm. This may not be the preferable term to use, however it would involve translating the application for implementation on a single massively parallel machine. It would not necessarily need to be run over thousands of pUnits; however the minor latencies involved in shared memory SIMD systems would be a great boon to performance. Furthermore, the removal of the need for an explicit master-slave relationship could greatly benefit performance by removing the need for a single pUnit to devote its time to organising many other pUnits. Drawbacks of this approach are additional complexity in implementation and cost of hardware.

8. References:

Alberts, B, Bray, D, Lewis, J, Raff, M, Roberts, K & Watson, JD 2002, *Molecular biology of the cell*, 4th edn, New York, Garland Science.

Amdahl, GM 1967, 'Validity of the single processor approach to achieving large scale computing capabilities', *AFIPS Conference Proceedings*, vol. 30, no. 8, pp. 483-485.

Anderson, DP, Cobb, J, Korpela, E, Lebofsky, M & Werthimer, D 2002, 'SETI@ home: an experiment in public-resource computing', *Communications of the ACM*, vol. 45, no. 11, pp. 56-61.

Baxevanis, AD & Ouellette, BFF 2001, *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*, Wiley-Interscience.

Brutlag, DL, Dautricourt, J-P, Diaz, R, Fier, J, Moxon, B & Stamm, R 1993, 'BLAZE(TM): An implementation of the Smith-Waterman sequence comparison algorithm on a massively parallel computer', *Computers & Chemistry*, vol. 17, no. 2, pp. 203-207.

Campbell, Reece & Meyers 2006, *Biology Seventh Edition, Australian Version*, Pearson, Sydney, NSW.

Carrillo, H & Lipman, D 2006, 'The Multiple Sequence Alignment Problem in Biology', *SIAM Journal on Applied Mathematics*, vol. 48, no. 5, pp. 1073-1082.

Collins, JF & Coulson, AFW 1984, 'Applications of parallel processing algorithms for DNA sequence analysis', *Nucl. Acids Res.*, vol. 12, no. 1Part1, pp. 181-192.

Dayhoff, MO 1972, *Atlas of protein sequence and structure*, vol. 5, National Biomedical Research Foundation Washington, Washington D.C.

Durbin, R, Krogh, A, Mitchison, G & Eddy, SR 1998, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press.

Ebedes, J & Datta, A 2004, 'Multiple sequence alignment in parallel on a workstation cluster', *Bioinformatics*, vol. 20, no. 7, pp. 1193-1195.

Feitelson, G & Treinin, M 2002, 'The blueprint for life?' *Computer*, vol. 35, no. 7, pp. 34-40.

George, DG, Barker, WC & Hunt, LT 1990, 'Mutation data matrix and its uses', *Methods Enzymol*, vol. 183, pp. 333-351.

Gonnet, GH, Cohen, MA & Benner, SA 1992, 'Exhaustive matching of the entire protein sequence database', *Science*, vol. 256, no. 5062, pp. 1443-1445.

Gotoh, O 1982, 'An improved algorithm for matching biological sequences', *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705-708.

---- 1996, 'Significant improvement in accuracy of multiple protein sequence alignments by iterative refinement as assessed by reference to structural alignments', *J. Mol. Biol*, vol. 264, no. 4, pp. 823-838.

Gupta, SK, Kececioglu, JD & Schaeffer, AA 1995, 'Improving the Practical Space and Time Efficiency of the Shortest-Paths Approach to Sum-of-Pairs Multiple Sequence Alignment', *Journal of Computational Biology*, vol. 2, no. 3, pp. 459-472.

Gusfield, D 1997, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Pr.

Henikoff, S & Henikoff, JG 1992, 'Amino Acid Substitution Matrices from Protein Blocks', *Proceedings of the National Academy of Sciences*, vol. 89, no. 22, pp. 10915-10919.

Hirschberg, DS 1975, 'A linear space algorithm for computing maximal common subsequences', *Communications of the ACM*, vol. 18, no. 6, pp. 341-343.

Holland, J 1959, 'A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously', paper presented to East Joint Computer Conference.

ISI 1981, *RFC 793 Transmission Control Protocol (v4)*, viewed 28th October 2007, <http://tools.ietf.org/html/rfc793>.

Julie D. Thompson, PKRROP 2005, 'BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark', *Proteins: Structure, Function, and Bioinformatics*, vol. 61, no. 1, pp. 127-136.

Kececioglu, J & Starrett, D 2004, 'Aligning alignments exactly', *Proceedings of the eighth annual international conference on Computational molecular biology*, pp. 85-96.

Kececioglu, J & Zhang, W 1998, 'Aligning Alignments', *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, vol. Spring-Verlag Lecture Notes in Computer Science 1448, pp. 189-208.

Kececioglu, JD 1983, 'The Maximum Weight Trace Problem in Multiple Sequence Alignment', *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, pp. 106-119.

King, RC & Stansfield, WD 1997, *A dictionary of genetics*, Oxford University Press, Oxford.

Kingston 2003, *Intel Dual-Channel DDR Memory Architecture White paper*, Kingston, viewed 28th October 2007, <http://www.kingston.com/newtech/MKF_520DDRwhitepaper.pdf>.

Li, K-B 2003, 'ClustalW-MPI: ClustalW analysis using distributed and parallel computing', *Bioinformatics*, vol. 19, no. 12, pp. 1585-1586.

Martins, WS, del Cuvillo, JB, Useche, FJ, Theobald, KB & Gao, GR 2001, 'A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison', *Pacific Symposium on Biocomputing*, vol. 6, pp. 311-322.

Mattson, TG, Sanders, BA & Massingill, BL 2005, *Patterns for parallel programming*, Addison-Wesley Boston.

McClure, MA, Vasi, TK & Fitch, WM 1994, 'Comparative analysis of multiple protein-sequence alignment methods [published erratum appears in Mol Biol Evol 1994 Sep;11(5):811]', *Mol Biol Evol*, vol. 11, no. 4, pp. 571-592.

Microsoft 2007a, *BinaryFormatter Class*, Microsoft, 29/10/2007, <http://msdn2.microsoft.com/en-us/library/system.runtime.serialization.formatters.binary.binaryformatter(VS.80).aspx>.

---- 2007b, *Stopwatch Class*, Microsoft, 25/10/2007, <http://msdn2.microsoft.com/en-us/library/system.diagnostics.stopwatch(vs.80).aspx>.

Morgenstern, B, Frech, K, Dress, A & Werner, T 1998, 'DIALIGN: finding local similarities by multiple sequence alignment', *Bioinformatics*, vol. 14, no. 3, pp. 290-294.

Mount, DW 2001, *Bioinformatics: sequence and genome analysis*, Cold Spring Harbor Laboratory Press, Cold Spring Harbor, N.Y.

Myers, EW & Miller, W 1988, 'Optimal alignments in linear space', *Comput. Appl. Biosci.*, vol. 4, no. 1, pp. 11-17.

Needleman, SB & Wunsch, CD 1970, 'A general method applicable to the search for similarities in the amino acid sequence of two proteins', *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443-453.

Notredame, C 2002, 'Recent progress in multiple sequence alignment: a survey', *Pharmacogenomics*, vol. 3, no. 1, pp. 131-144.

Notredame, C, Higgins, DG & Heringa, J 2000, 'T-Coffee: A novel method for fast and accurate multiple sequence alignment', *J. Mol. Biol*, vol. 302, no. 1, pp. 205-217.

OED 1989, *The Oxford English Dictionary 2nd Edition. OED Online.*, Oxford University Press, <http://dictionary.oed.com/cgi/entry/50178241>.

Pevzner, PA 2000, *Computational molecular biology: An Algorithmic Approach*, MIT Press, Cambridge, Mass.

Schmollinger, M, Nieselt, K, Kaufmann, M & Morgenstern, B 2004, 'DIALIGN P: Fast pair-wise and multiple sequence alignment using parallel processors', *BMC Bioinformatics*, vol. 5, no. 1, p. 128.

Schwartz, S, Miller, W, Yang, CM & Hardison, RC 1991, 'Software tools for analyzing pairwise alignments of long sequences', *Nucleic Acids Res*, vol. 19, no. 17, pp. 4663-4667.

Smith, TF & Waterman, MS 1981, 'Identification of Common Molecular Subsequences', *J. Mol. Bwl*, vol. 147, pp. 195-197.

Thompson J. D. , Koehl P., Ripp R. & Poch O. 2005, 'BAliBASE 3.0: Latest developments of the multiple sequence alignment benchmark', *Proteins: Structure, Function, and Bioinformatics*, vol. 61, no. 1, pp. 127-136.

Thompson, JD, Higgins, DG & Gibson, TJ 1994, 'CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice', *Nucl. Acids Res.*, vol. 22, no. 22, pp. 4673-4680.

Thompson, JD, Plewniak, F & Poch, O 1999, 'BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs', *Bioinformatics*, vol. 15, no. 1, pp. 87-88.

Trelles-Salazar, O, Zapata, EL & Carazo, JM 1994, 'On an efficient parallelization of exhaustive sequence comparison algorithms on message passing architectures', *Comput. Appl. Biosci.*, vol. 10, no. 5, pp. 509-511.

Wang, L & Jiang, T 1994, 'On the complexity of multiple sequence alignment', *J Comput Biol*, vol. 1, no. 4, pp. 337-348.

Watson, JD & Crick, FHC 1953, 'Molecular Structure Of Nucleic Acids A Structure for Deoxyribose Nucleic Acid', *Nature*, vol. 171, no. 4356, pp. 737-738.

Wilkinson, B & Allen, M 1998, *Parallel programming: techniques and applications using networked workstations and parallel computers*, Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

Yap, TK, Frieder, O & Martino, RL 1998, 'Parallel computation in biological sequence analysis', *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 3, pp. 283-294.

Zomaya, AY 1996, *Parallel and Distributed Computing Handbook*, McGraw-Hill Professional, New York.

9. APPENDICES

This section describes the availability of tools used during research

.

BAliBASE dataset: See associated CD-ROM.

AAE serial implementation: See associated CD-ROM.

AAE Master/Slave implementation: See associated CD-ROM.

BLOSUM45 cost matrix: See associated CD-ROM.

Results: See associated CD-ROM.

ClustalW2: See http://www.ebi.ac.uk/Tools/clustalw2/ (Last accessed 2/11/2007).