University of Alberta

Library Release Form

Name of Author: Adrian Radu Driga

Title of Thesis: Parallel FastLSA: A Parallel Algorithm for Pairwise Sequence Alignment

Degree: Master of Science

Year this Degree Granted: 2002

. . . .

Adrian Radu Driga
Apt. 201, 10745 - 83 Ave
Edmonton, Alberta
Canada, T6E 2E5

Date: . . . .

University of Alberta

Parallel FastLSA: A Parallel Algorithm for Pairwise Sequence Alignment

by

Adrian Radu Driga

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Parallel FastLSA: A Parallel Algorithm for Pairwise Sequence Alignment** submitted by Adrian Radu Driga in partial fulfillment of the requirements for the degree of **Master of Science**.

. . . .
Paul Lu

. . . .
Duane Szafron

. . . .
Warren Gallin

Date: . . . .

Abstract

We introduce a novel parallel algorithm for optimal pairwise sequence alignment, called *Parallel FastLSA*. *Parallel FastLSA* is the parallel version of an existing algorithm, *FastLSA*, whose name is an abbreviation for *Fast Linear Space Alignment*. *FastLSA* finds an optimal alignment for two biological sequences using linear space, and it is proven to be empirically faster than two other frequently used alignment algorithms: *Needleman–Wunsch* and *Hirschberg*. *Parallel FastLSA* is designed to further improve the time performance of *FastLSA*, while still using only linear space.

We describe in detail the *Parallel FastLSA* algorithm and how the algorithm is implemented. We also analyze the effectiveness of *Parallel FastLSA* and give detailed accounts of its theoretical and empirical performance. Our experimental results show that *Parallel FastLSA* exhibits good speedups, almost linear for 8 processors or less, and also that the efficiency of *Parallel FastLSA* increases with the size of the sequences that are aligned.

Acknowledgements

Many thanks to

- Paul Lu for being a great supervisor and for the help provided throughout my Master's program;
- Jonathan Schaeffer and Duane Szafron for the insigthful discussions on sequence alignment and *FastLSA*;
- MACI and the Research Support Group at CNS for making their facilities and expertise available for my research;
- Anne Nield for proof-reading this thesis;
- University of Alberta and the Department of Computing Science for giving me the opportunity to study under their banners.

Contents

List of Tables

List of Figures

List of Symbols

- m, n lengths of sequences
- k number of Grid Cache rows and columns
- BM size of Base Case buffer
- RM number of memory units available
- P number of processors
- u number of rows of tiles between consecutive Grid Cache rows
- v number of columns of tiles between consecutive Grid Cache columns
- R total number of rows of tiles
- C total number of columns of tiles
- $optimal \binom{x}{y}$ an optimal alignment between the sequences x and y
- $\Leftrightarrow$ the equivalence relation

List of Abbreviations

- **BLAST** Basic Local Alignment Search Tool
- **bp** base pair
- **ccNUMA** cache-coherent nonuniform memory access
- **DCBDS** Divide & Conquer Bidirectional Search
- **DCFS** Divide-and-Conquer Frontier Search
- **DNA** deoxyribonucleic acid
- **d.p. matrix** dynamic programming matrix
- **FastLSA** Fast Linear Sequence Alignment
- **PSI-BLAST** Position-Specific Iterated BLAST
- **RNA** ribonucleic acid

Chapter 1

Introduction

We introduce a novel parallel algorithm for optimal pairwise sequence alignment, called *Parallel FastLSA*. This algorithm is the parallel version of an existing algorithm, *FastLSA* [Charter *et al.*, 2000], whose name is a contracted form for *Fast Linear Space Alignment*. As its name suggests, *FastLSA* finds an optimal alignment for two biological sequences using linear space, and it is proven to be empirically faster than two other frequently used alignment algorithms: *Needleman–Wunsch* [Needleman and Wunsch, 1970] and *Hirschberg* [Hirschberg, 1975]. *Parallel FastLSA* aims to further improve the time performance of *FastLSA*, while still using linear space. Our experimental results show that *Parallel FastLSA* exhibits good speedups, and its efficiency increases with the size of the sequences that are aligned.

This chapter introduces the basic concepts of molecular biology that are required in order to understand the biological background of the thesis. Because this is a thesis in computing science, the description of the biological concepts is not detailed to the extent that would be necessary in the field of molecular biology. Also, this introduction covers neither the mechanisms of molecular genetics nor the techniques used for studying genomes. A molecular biology textbook (e.g., [Campbell, 1999]) can provide a complete and insightful picture concerning these subjects.

1.1 Notions of Biochemistry

Modern research has shown that all living organisms have a similar molecular chemistry. The most important molecules involved in bio–chemistry are *proteins* and *nucleic acids*. Proteins are responsible for the physical life of an organism through regulating its metabolism. Nucleic acids encode the information necessary to produce proteins, and are also responsible for passing this information to subsequent generations.

Research in molecular biology is mainly devoted to the study of the structure and function of proteins and nucleic acids. The remainder of this section gives brief descriptions of these important actors in the chemistry of life.

	One-letter code	**Three-letter code**	**Name**
1	A	Ala	Alanine
2	C	Cys	Cysteine
3	D	Asp	Aspartic Acid
4	E	Glu	Glutamic Acid
5	F	Phe	Phenylalanine
6	G	Gly	Glycine
7	H	His	Histidine
8	I	Ile	Isoleucine
9	K	Lys	Lysine
10	L	Leu	Leucine
11	M	Met	Methionine
12	N	Asn	Aspargine
13	P	Pro	Proline
14	Q	Gln	Glutamine
15	R	Arg	Arginine
16	S	Ser	Serine
17	T	Thr	Threonine
18	V	Val	Valine
19	W	Trp	Tryptophan
20	Y	Tyr	Tyrosine

Table 1.1: The Twenty Amino Acids Commonly Found in Proteins

1.1.1 Proteins

A protein is a macromolecule formed by chaining together simpler molecules called *amino acids.* Table 1.1 lists the most common 20 amino acids found in proteins. The protein molecules are represented as strings of letters over the twenty-letter alphabet created by the amino-acid letters.

Proteins constitute the majority of the substance types present in living organisms. For example, *structural proteins* are the building blocks of tissues, while *enzymes* act as catalysts for biochemical reactions. Other proteins are used for the transport of oxygen, or act as antibodies for the immune system.

To be precise, it should be mentioned that only *residues* of the original amino acids are present in the molecular chain. For this reason, the length of a protein is measured in *residues* rather than in *amino acids*, although often the latter term is used. Typical proteins are 300-residues long [Setubal and Meidanis, 1997], but there are proteins with as few as 100 or as many as 5,000 residues.

1.1.2 Nucleic Acids

There are two different types of nucleic acids: *deoxyribonucleic acid* (*DNA*) and *ribonucleic acid* (*RNA*). Similar to the proteins, DNA molecules are formed by the chaining together of simpler molecules. DNA, however, contains two such chains, referred to as *strands.* The molecules that make up each strand are called *nucleotides.* A DNA nucleotide consists of a deoxyribose, a phosphate residue, and a nitrogenated *base*. There are four types of DNA nucleotides corresponding to four distinct bases: adenine (A), guanine (G), cytosine (C), and thymine (T). Bases A and G belong to the group of *purines*, while C and T belong to the group of *pyrimidines*. A single-stranded DNA sequence has a canonical direction, and its length is measured in *bases* or *nucleotides*.

DNA molecules are much longer than the proteins. For example, the DNA molecules can be millions of nucleotides long, while the proteins can be at most thousands of residues long. It should also be noted that each amino acid is encoded by triplets of nucleotides of DNA, and these triplets are called *codons* [Waterman, 1995].

The two strands of a DNA molecule are tied together in a helical structure. This is the famous double helix structure discovered by James Watson and Francis Crick in 1953. The structure holds because each base in one strand bonds to a base in the other strand. Pairs are always formed between the bases A and T, and between G and C. These pairs are known as *Watson-Crick base pairs*, and their bases are referred to as *complementary bases*. Most often, base pairs (*bp*) are used as the unit of length for the DNA molecules.

RNA molecules are similar to the DNA molecules, with small differences in composition and structure. In RNA, the sugar is ribose instead of deoxyribose, and uracil (U) bases are present instead of thymine (T). Like thymine (T), uracil (U) binds with adenine (A). The most significant difference is that RNA does not form a double helix, although hybrid RNA-DNA helices are frequent. Also, parts of an RNA molecule may bind to other parts of the same molecule through complementarity. In terms of functionality, while DNA is used only for encoding information, there are multiple types of RNAs in a cell, performing different expression functions.

Both types of nucleic acids, DNA and RNA, are represented as strings of letters over the four-letter alphabet created by the letters corresponding to the nucleotides from which they are made.

1.2 Biological Sequence Alignment – Motivation

Since the structure of DNA was uncovered in 1953, molecular biology research has advanced tremendously. The techniques used in biological laboratories have improved considerably, allowing a large amount of data to be generated. The size and complexity of the data make their manipulation a challenging task. Some of the problems that appear in the process of studying the collected data belong to the fields of mathematical and computing sciences. For example, databases are used to store all the information that is generated. Complex statistical knowledge is required for the efficient querying of these databases. Understanding the molecular sequences requires algorithms and pattern recognition expertise. The need for efficient solutions for these problems led to the emergence of a new scientific field: *computational molecular biology*, also known as *bioinformatics*.

The topic of this thesis is "the most important primitive operation in computational molecular biology" [Setubal and Meidanis, 1997]: *sequence comparison.* Informally stated, sequence comparison gives the degree of *similarity* between two biomolecular sequences. Comparison tests between two sequences are performed in a variety of situations, some of which are listed below. Furthermore, sequence comparison serves as a basis for other complex manipulations of the biological sequences (e.g., multiple sequence alignment).

The result of the sequence comparison operation applied to a pair of biological sequences can be visualized as the *alignment* of the two sequences. The alignment is obtained by putting one sequence above the other in order to emphasize the correspondence between similar characters or substrings of the two sequences.

Setubal and Meidanis [Setubal and Meidanis, 1997] identify a list of problems that are related to sequence comparison and appear frequently in computational biology research. A summary of these problems follows:

1. Two sequences over the same alphabet are almost equal, except for a few isolated insertions, deletions, and substitutions of characters. The average frequency of these differences is very low; however, their exact positions must be found. This problem occurs when a gene is sequenced by two different labs and the results are compared.

2. Two sequences over the same alphabet, with a few hundred characters each, are given. The problem is to decide whether there is a prefix of one sequence that is similar to a suffix of the other. If the answer is yes, the matching prefix and suffix must be produced. This problem appears when small DNA fragments are assembled into a longer sequence in the process of large-scale DNA sequencing. Often, this problem must be applied pairwise to several hundred sequences, most of which are unrelated.

3. Two sequences over the same alphabet, with a few hundred characters each, are given. The problem is to decide whether there are two substrings, one from each sequence, that are similar. This problem appears when searching for local similarities in large sequence databases. In this context, a sequence must be compared against thousands of others.

All these problems can be solved using the same basic algorithmic idea that is used for solving the sequence comparison problem. The following section gives the formal details of the pairwise sequence alignment problem. Computers are needed to perform the sequence comparison operation because enormous amounts of data are involved, especially when querying biosequence databases. Using a computer to do these operations is less error-prone and more convenient.

1.3 Aligning Two Sequences

Consider the following pair of DNA sequences: `ATAGTC` and `ATTAGGC`. At a glance, they look very much alike, and this becomes more obvious when they are aligned one above the other:

```
A-TAGTC
ATTAGGC
```

The only differences are an extra T in the second sequence and a change from T to G in the second to last position. Note that a gap, marked with a "-", is introduced in the first sequence in order to allow the bases before and after the gap to align perfectly. This is a sample alignment of the two sequences.

Formally, an alignment of two sequences is obtained from the original sequences by inserting gaps until the resulting sequences are of the same size. An alignment also must obey the restriction that gaps cannot appear in the same position in both sequences. The example above satisfies the definition of an alignment.

The goal of the sequence comparison operation is to find an optimal alignment of two sequences relative to a *cost function*. One type of cost function is obtained by assigning a *score* to an alignment in the following manner: each column of the alignment is given a value based on the two characters forming the column, and the *total score* of the alignment is the sum of all the values assigned to its columns. If a column has two identical characters, it is valued with +2 (i.e., a *match*). Different characters are valued with −1 (i.e., a *mismatch*). Finally, if a gap is present, the column is valued with −2 (i.e., a *gap penalty*). An optimal alignment is one with maximal total score among the total scores of all possible alignments between the two sequences. In general, there may be many optimal alignments between two sequences.

For the alignment in the example above, there are five columns with identical characters, one column with distinct characters, and one column with a gap, giving a total score of

$$5 \times 2 + 1 \times (-1) + 1 \times (-2) = 7 \tag{1.1}$$

The particular values +2, −1, and −2 were chosen because they constitute a simple implementation of the policy of rewarding matches, and penalizing mismatches and gaps. In practice, the value which is assigned to a column depends on the probability with which the character from the first row can transform itself into the character from the second row after a certain number of evolutionary steps. With regard to proteins, the amino acids have biochemical properties that influence the way they replace each other during the evolution of a protein [Setubal and Meidanis, 1997]. For example, it is more likely that amino acids of similar sizes will be substituted for one another than those of widely different sizes. The tendency to bind with water molecules also influences the probability of mutual substitution. Because protein comparisons are usually performed to establish an evolutionary relation between sequences, it is important to use scoring functions that reflect these probabilities accurately.

Often, the best method to derive similarity scores for pairs of residues is to empirically observe the actual substitution rates; doing so is advisable because it is difficult to account for all the factors that influence the probability of mutual substitution of amino acids. A standard procedure for achieving this goal is based on an important family of scoring matrices, known as the PAM matrices. The acronym PAM stands for *Point Accepted Mutations*, or *Percent of Accepted Mutations*, a reference to the fact that "the basic PAM-1 matrix reflects an amount of evolution producing on average one mutation per hundred amino acids" [Setubal and Meidanis, 1997]. The PAM matrices were introduced by Dayhoff *et al.* [Dayhoff *et al.*, 1978].

Before two sequences are aligned, the evolutionary distance at which to compare them must be chosen. The PAM matrices are functions of this distance. For instance, a PAM-250 matrix is suitable for comparing sequences that are 250 units of evolution apart. If no information on the true evolutionary distance between the two sequences is available, the recommended approach is to align the sequences using several PAM matrices that cover a wide range – for example, PAM-40, PAM-120, and PAM-250. In general, low PAM numbers are good for finding short, strong, local similarities, while high PAM numbers detect long, weak ones [Setubal and Meidanis, 1997]. It should also be noted that PAM matrices consider the mutations at the amino acid level only, without involving the DNA level.

1.4 Thesis Contributions

The main contribution of this thesis is a new parallel algorithm for optimal pairwise sequence alignment, called *Parallel FastLSA*. We developed and implemented *Parallel FastLSA* and ran extensive experiments with the algorithm. The thesis describes in detail the algorithm and how it can be implemented. We also give detailed accounts of the theoretical and empirical performance of *Parallel FastLSA*.

Parallel FastLSA is the parallel version of an existing algorithm, *FastLSA* [Charter *et al.*, 2000]. *FastLSA* finds an optimal alignment for two biological sequences using linear space, and it is shown [Charter *et al.*, 2002] to be empirically faster than two other frequently used alignment algorithms: *Needleman–Wunsch* [Needleman and Wunsch, 1970] and *Hirschberg* [Hirschberg, 1975]. *Parallel FastLSA* aims to further improve the time performance of *FastLSA*, while still using linear space. The space linearity of *Parallel FastLSA* and its quadratic time complexity are guaranteed by two theoretical results that we state and prove.

Our experiments show that *Parallel FastLSA* exhibits good speedups, and its efficiency increases with the size of the sequences that are aligned. For example, the speedup on 16 processors is 9.03 for the alignment of two sequences of size 37,785 bp and 37,349 bp. The speedup then increases to 10.83 for the alignment of two sequences of size 55,820 bp and 66,315 bp. When two much longer sequences of length 319,030 bp and 305,636 bp are aligned, the speedup for 16 processors goes up to 13.62. The remainder of this section is an overview of the material presented in this thesis.

Following this introductory chapter, Chapter 2 focuses on the existing work related to the sequence alignment problem. We discuss the algorithms of *Needleman–Wunsch* and *Hirschberg* because they are the most widely used algorithms for optimal pairwise sequence alignment. *FastLSA* was developed as a generalization of the *Hirschberg* algorithm and, because it is the base for *Parallel FastLSA*, it is also discussed in Chapter 2. These three sequential algorithms for sequence alignment are described in detail, and a commentary on their space and time requirements follows the description of each algorithm.

Chapter 3 introduces the *Parallel FastLSA* algorithm. We start with a description of the parallel algorithm and then give further details about the two versions of *Parallel FastLSA* that we have implemented. The last section of this chapter gives a detailed account of the theoretical space and time complexity of *Parallel FastLSA*.

The empirical analysis of *Parallel FastLSA* is presented in Chapter 4. We explain the setup for the experiments that we performed with *Parallel FastLSA* and comment extensively on the results of the benchmarks that we ran. For each of the three pairs of real life sequences that we align using *Parallel FastLSA*, we present performance graphs that show consistently the effectiveness of the algorithm.

Chapter 5 lists the three most important aspects of *Parallel FastLSA* that we would like to investigate as future work. We conclude this chapter and the thesis with a short summary of the contributions and the results presented in this thesis.

1.5 Concluding Remarks

In this chapter we introduce the pairwise sequence alignment operation, and the most relevant notions of bio-chemistry related to it. We argue the importance of this operation and the importance of having an efficient algorithm to perform it. We also present an example of pairwise alignment of two sequences and briefly introduce the framework in which the alignment operation will be discussed in the following chapters. This chapter also features a summary of the contributions of the thesis and an overview of the work presented.

Chapter 2

Related Work

A naïve approach for finding the best alignment between two biological sequences would be to generate all the possible alignments between the two sequences and then pick the best one, relative to the specified cost function. However, because the number of possible alignments between the two sequences is exponential in the size of the sequences, the resulting algorithm would be awkwardly slow. Fortunately, more efficient algorithms exist, and they are described in the following sections. Because the main goal of this thesis is to introduce a new, faster algorithm for sequence alignment, emphasis is placed on explaining the space and time complexity of each algorithm presented.

2.1 The ***Needleman–Wunsch*** Algorithm

The *Needleman–Wunsch* algorithm is based on *dynamic programming.* Given two sequences, a and b, the score for an optimal alignment between them is computed by considering the scores of the best alignments between their *prefixes* of arbitrary length. The algorithm starts by finding the score for an optimal alignment for short prefixes of the two sequences. It then uses these results to solve the problem for larger prefixes. After the score for an optimal alignment between a and b is found, one or all optimal alignments are built during a post-processing phase. Details of this algorithm, which is an example of a *full matrix* algorithm, follow.

Suppose that a is m characters long, and b is n characters long. The string a has $m+1$ prefixes denoted by $a[1..i]$, with i ranging from 0 to m. When $i=0$, $a[1..0]$ denotes the empty prefix of a. Similarly, the $n+1$ prefixes of b are denoted by $b[1..j]$, with j ranging from 0 to n. *Needleman–Wunsch*'s algorithm computes the score for an optimal alignment of all pairs of prefixes of a and b. The score for an optimal alignment between $a[1..i]$ and $b[1..j]$ is stored in the position (i,j) of the $(m+1)\times(n+1)$ matrix S. The value of the entry (i,j) in the matrix S is referred to as $S[i,j]$. The goal of the first phase of the algorithm is to compute $S[m,n]$. To achieve this goal, all the entries of the matrix S must be computed.

The dynamic programming paradigm is suitable for solving the sequence alignment problem across a wide range of cost function types, including the match/mismatch and PAM matrix cost functions discussed in Section 1.3. Suppose that the sequences a and b are to be aligned using a scoring system that penalizes a gap with *Gap_Penalty*, usually a negative number, and adds $align(X, Y)$ to the alignment score when a column of the alignment consists of the characters X and Y. An example of possible values for *Gap_Penalty* and $align(X, Y)$ is shown below, but for now consider them as literal variables. For this scoring system, *Needleman–Wunsch*'s algorithm initializes the first row and the first column of the matrix S with multiples of *Gap_Penalty*, i.e.,

$$S[i, 0] = i \times \mathit{Gap\_Penalty}, \forall i, 0 \leq i \leq m \tag{2.1}$$

$$S[0, j] = j \times \mathit{Gap\_Penalty}, \forall j, 0 \leq j \leq n \tag{2.2}$$

This is consistent with the definition of $S[i, 0]$ and $S[0, j]$ because there is only one possible alignment if one of the sequences is empty: a gap is to be added for each character in the other sequence.

In order to compute the other entries of the dynamic programming (*d.p.*) matrix S, the algorithm takes advantage of the fact that an alignment for $a[1..i]$ and $b[1..j]$ is obtained only through one of the following three methods:

- align $a[1..i-1]$ with $b[1..j-1]$ and align $a[i]$ with $b[j]$, or
- align $a[1..i]$ with $b[1..j-1]$ and align a gap with $b[j]$, or
- align $a[1..i-1]$ with $b[1..j]$ and align $a[i]$ with a gap.

These possibilities are exhaustive because the last column of the alignment cannot contain two gaps. As a consequence, the score for an optimal alignment between $a[1..i]$ and $b[1..j]$ can be determined using the formula:

$$S[i, j] = \max \begin{cases} S[i-1, j-1] + align(a[i], b[j]) \\ S[i, j-1] + \mathit{Gap\_Penalty} \\ S[i-1, j] + \mathit{Gap\_Penalty}. \end{cases} \tag{2.3}$$

It should be noted that the value of the entry (i, j) depends only on the values of the entries $(i-1, j-1)$, $(i, j-1)$, and $(i-1, j)$. Therefore, these last three values must be available when $S[i, j]$ is computed. For example, this can be done by computing the d.p. matrix S row by row, and left to right on each row.

Figure 2.1 shows the dynamic programming matrix S corresponding to $a = \texttt{ATAGTC}$ and $b = \texttt{ATTAGGC}$. Sequence a is placed vertically along the left margin of the matrix, while b is placed horizontally along the top. The entries of S are computed using the scoring function

$$\mathit{Gap\_Penalty} = -2, \tag{2.4}$$

and

$$align(X, Y) = \begin{cases} +2 & \text{if } X = Y (match) \\ -1 & \text{if } X \neq Y (mismatch), \end{cases} \tag{2.5}$$

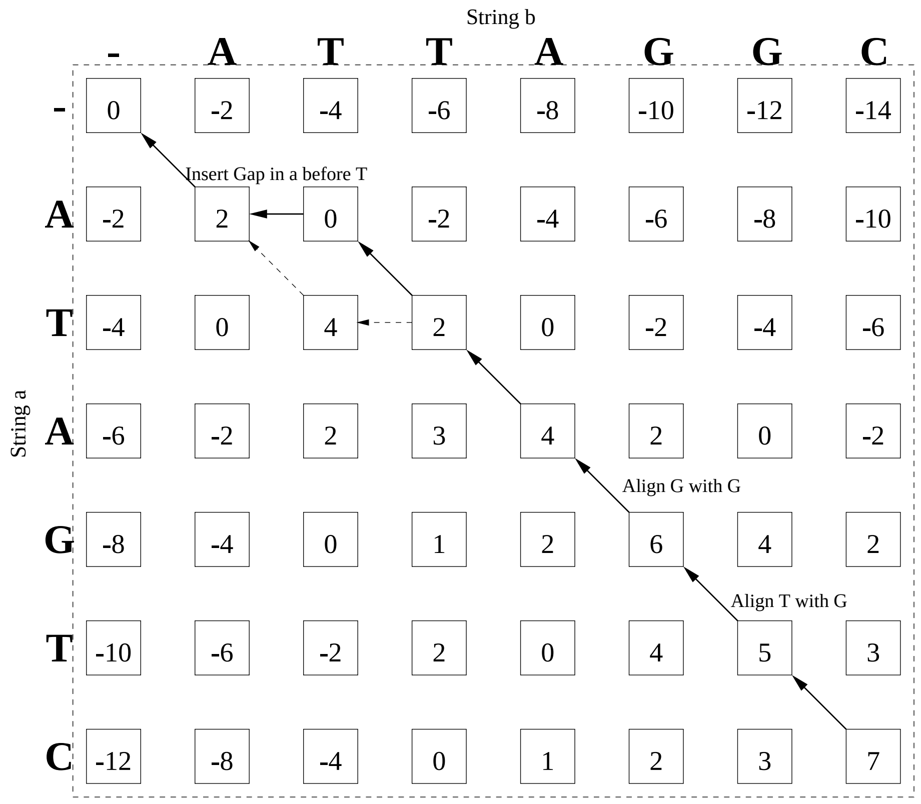


Figure 2.1: Optimal Paths through the Full Matrix for `ATAGTC` and `ATTAGGC`

which is the scoring function introduced in Section 1.3. The score for an optimal alignment between a and b is given by the entry $(m, n) = (6, 7)$ in the matrix S. Its value, 7, is equal to that computed by Formula 1.1 in Section 1.3.

Needleman–Wunsch's algorithm not only produces the score for an optimal alignment of a and b, but also gives the opportunity to find one or all optimal alignments between the two sequences. The idea is to build a path through S that starts at (m, n), ends at $(0, 0)$, and satisfies the following property: if (i, j) belongs to the path, then an entry that supplied the maximum value in Formula 2.3 also belongs to the path. Such a path is called an *optimal path* because it corresponds uniquely to an optimal alignment between a and b. Given an optimal path, the corresponding optimal alignment is built following the rules:

- if (i, j) - $(i - 1, j - 1)$ is an arc on the optimal path (diagonal arrow), then $a[i]$ is aligned with $b[j]$;

- if (i, j) - $(i, j - 1)$ is an arc on the optimal path (horizontal arrow), then "-" is aligned with $b[j]$;

- if (i, j) - $(i - 1, j)$ is an arc on the optimal path (vertical arrow), then $a[i]$ is aligned with "-".

An optimal path is built by applying a procedure that traces back an optimal path after the d.p. matrix is filled. If the links between each entry in S and the neighboring entries that supplied the maximum for its computation are stored during the filling of S, this traceback procedure simply builds a path from (m, n) to $(0, 0)$ following the links. Otherwise, the traceback procedure starts with (m, n) and finds the next entry in the path by applying a test based on Formula 2.3. Note that the maximum value in Formula 2.3 can be reached by more than one branch. This situation leads to multiple optimal paths in the dynamic programming matrix. Through the use of a stack, the traceback procedure can produce all optimal paths.

Figure 2.1 shows two distinct optimal paths through the d.p. matrix S. The continuous path corresponds to the alignment

```
A-TAGTC
ATTAGGC
```

and the alternate path corresponds to the alignment

```
AT-AGTC
ATTAGGC
```

These are the only paths that are optimal for the strings $a =$ `ATAGTC` and $b =$ `ATTAGGC`.

Figure 2.2 presents the algorithm that computes the d.p. matrix S for two given sequences, a and b. This algorithm also fills the matrix of links L, which is used to produce an optimal alignment of a and b. Specifically, $L[i, j]$ points to one of the positions $(i-1, j-1)$, $(i, j-1)$ or $(i-1, j)$ depending on which entry supplied the maximum in Formula 2.3. Note that the pseudo-code uses the construct (i, j) as a reference to the position of $S[i, j]$ in the matrix S. Function *max_supply* in Figure 2.2 determines which of the three adjacent entries has supplied the maximum for the computation of $S[i, j]$. In case of a tie, the order in which the positions are listed above dictates their precedence. As a result, the optimal alignment returned by this algorithm has the characteristic that whenever there is a tie, a column with two letters has precedence over a column with a gap in a, which in turn has precedence over a column with a gap in b.

The *Needleman–Wunsch* algorithm shown in Figure 2.2 returns an optimal alignment in the pair of global vectors *align_a* and *align_b*. These vectors contain the *len* aligned characters, which can be either gaps or letters from the sequences. The optimal score is also returned.

It is easy to analyze the time and space complexity of the algorithm *Needleman–Wunsch*. The first two loops are initialization loops which consume $O(m)$ and $O(n)$ time. The two nested loops of the algorithm fill the dynamic programming matrix in a time proportional to the size of the matrix, i.e., $O(mn)$. The construction of the alignment in the last loop is done in $O(len)$ time, where len is the size of the returned alignment. Because $\max(|a|, |b|) \leq len \leq m + n$, the last phase of the algorithm actually takes $O(m + n)$ time. Overall, the

```
Algorithm Needleman-Wunsch
    input: sequences a and b
    output: optimal alignment in align_a, align_b,
            and the score of the alignment

    m = |a|
    n = |b|
    S[0,0] = 0
    for i = 1 to m do
        S[i,0] = i x Gap_Penalty
        L[i,0] = (i-1,0)
    for j = 1 to n do
        S[0,j] = j x Gap_Penalty
        L[0,j] = (0,j-1)
    for i = 1 to m do
        for j = 1 to n do
            S[i,j] = max( S[i-1,j-1] + align(a[i],b[j]),
                          S[i, j-1] + Gap_Penalty,
                          S[i-1, j] + Gap_Penalty )
            L[i,j] = max_supply( (i-1,j-1), (i, j-1), (i-1, j) )
    i = m
    i = n
    len = 0
    while i > 0 or j > 0
        if L[i,j] == (i-1,j-1) then
            align_a[len] = a[i]
            align_b[len] = b[j]
        else if L[i,j] == (i,j-1) then
                 align_a[len] = '-'
                 align_b[len] = b[j]
             else
                 align_a[len] = a[i]
                 align_b[len] = '-'
        len = len + 1
        (i,j) = L[i,j]
    reverse( align_a )
    reverse( align_b )
    return S[m,n]
```

Figure 2.2: Pseudo-code for *Needleman–Wunsch*'s Algorithm

time complexity of the alignment is $O(mn)$ because this is the dominant term of the time complexity expression.

The space used by the algorithm is shared between the d.p. matrix space, $O(mn)$, and the alignment vectors space, $O(len) = O(m+n)$. Overall, the space complexity of the algorithm is $O(mn)$ because this is the dominant term of the space complexity expression. Hence, the complexity of the algorithm is $O(mn)$ for both time and space. When the input sequences are of equal length, n, the complexity of the algorithm is $O(n^2)$.

On a historical note, the paper by Needleman and Wunsch [Needleman and Wunsch, 1970] is generally accepted as the first important contribution in sequence comparison from the point of view of biologists. The algorithm described by Needleman and Wunsch has a fixed penalty for a gap, independent of its length. In this paper, no complexity analysis is given. Later analysis has shown that the algorithm, as presented, required cubic time to run [Setubal and Meidanis, 1997]. Today, quadratic time algorithms are available for a whole range of scoring functions, including *affine gap penalty* functions [Setubal and

Meidanis, 1997]. A constant function such as the *Gap_Penalty* used in this thesis is only a particular case of affine function. Despite the imperfections of the algorithm introduced in [Needleman and Wunsch, 1970], the name "*Needleman–Wunsch* algorithm" is often used to designate any kind of *global alignment* algorithm based on dynamic programming [Setubal and Meidanis, 1997].

A *global alignment* algorithm is one that finds an optimal alignment between the whole sequences a and b. The algorithm described in this section is for global alignment, and this thesis deals only with pairwise global alignment algorithms. Two other types of alignment are also popular with molecular biology researchers: *local alignment*, which is an alignment between a substring of a and a substring of b, and *semi-global alignment*, which is a global alignment that does not penalize the gaps inserted at the beginning or at the end of the aligned sequences. The goal of a local alignment algorithm is to find the highest scoring local alignment between the two sequences. Smith and Waterman [Smith and Waterman, 1981] were the first to produce a local alignment algorithm, leading to a situation where the name "*Smith–Waterman* algorithm" designates almost any local alignment algorithm based on dynamic programming. It should be noted that algorithms for finding the best local alignment, or the best semi-global alignment of two sequences, are small variations of the basic global alignment algorithms.

2.2 The *Hirschberg* Algorithm

The quadratic space complexity of the *Needleman–Wunsch* algorithm makes it unattractive for applications involving very long sequences. For example, aligning two DNA sequences of 100,000 base pairs requires at least 40 Gigabytes, assuming that each entry of the d.p. matrix is stored as a 4 Byte integer. This amount of memory is prohibitive for most commodity computers of today. Even if the dynamic programming matrix of this alignment fit in the main memory of a computer, the performance of the alignment operation would be inferior to the expected time of $O(10^{10})$ because of the cache misses the *Needleman–Wunsch* algorithm would incur. To date, no algorithm is known that uses asymptotically less time than $O(mn)$ and keeps the same generality as *Needleman–Wunsch*'s algorithm. However, with respect to space, the complexity can be improved from quadratic, $O(mn)$, to linear, $O(m+n)$, without losing any generality.

Hirschberg was the first to present a linear space algorithm capable of producing an optimal alignment of two sequences [Hirschberg, 1975]. Although Hirschberg made his observations in the context of the problem of finding a longest common subsequence of two strings, the results also apply to the sequence alignment problem. Myers and Miller are credited with developing the first linear space algorithm for optimal sequence alignment based on *Hirschberg*'s algorithm [Myers and Miller, 1988].

An important observation is that, in *Needleman–Wunsch*'s algorithm, the derivation of the row i requires only the row $i-1$ of the matrix to be known. Figure 2.3 presents an

```
Algorithm LastRow
    input: sequences a and b
    output: vector LL

    m = |a|
    n = |b|
    for j = 0 to n do
        LL[j] = j x Gap_Penalty
    for i = 1 to m do
        old = LL[0]
        LL[0] = i x Gap_Penalty
        for j = 1 to n do
            temp = LL[j]
            LL[j] = max( old + align(a[i],b[j]),
                         LL[j-1] + Gap_Penalty,
                         LL[j] + Gap_Penalty )
            old = temp
```

Figure 2.3: Pseudo-code for the *LastRow* Algorithm

algorithm that computes the score for an optimal alignment using linear space. Algorithm *LastRow* takes as input the strings $a[1..m]$ and $b[1..n]$, and produces as output the vector LL. This vector consists of the same values as the last row, m, of the dynamic programming matrix computed by the *Needleman–Wunsch* algorithm. However, the *LastRow* algorithm requires only $n+1$ locations of memory to compute the vector LL; hence, it is space linear in the size of the sequences.

The correctness of *LastRow* is supported by the following invariant assertions:

- at the beginning of step i of the outer loop, LL holds the values of the row $i-1$ of the d.p. matrix;
- in the inner loop, at the beginning of step j, $LL[0..j-1]$ holds the values of the row i, while $LL[j..n]$ holds the values of the row $i-1$;
- at step j, $old = S[i-1, j-1]$.

The computations done by the *Needleman–Wunsch* algorithm are mimicked using only the vector LL, and two temporary variables. Because of the two loops, the time complexity of the *LastRow* algorithm is $O(mn)$.

LastRow computes the similarity of the two sequences in $LL[n]$. However, finding an optimal alignment of the two sequences is not trivial since the d.p. matrix used in *Needleman–Wunsch*'s algorithm to retrieve an alignment is not available here. The key result of Hirschberg [Hirschberg, 1975] is a linear space algorithm which accepts as input the sequences a and b, and outputs an optimal alignment of them. The algorithm *LinearSpace*, which is shown in Figure 2.4, is not the original algorithm, but a version customized for the sequence alignment problem [Setubal and Meidanis, 1997].

LinearSpace is based on the *divide and conquer* paradigm. The vertical sequence a is split into two subsequences at the index $i = \lfloor m/2 \rfloor$. The goal is to find a column *pos* of the d.p. matrix such that the entry (i, pos) belongs to an optimal path. The fact that

```
Algorithm LinearSpace
    input: sequences a and b, indices r, s, u, v, and start
    output: optimal alignment between a[r..s] and b[u..v]
            placed in align_a and align_b,
            begins at position start,
            ends at position end

    if a[r..s] empty or b[u.v] empty then
        // BASE CASE
        Align the non-empty sequence with spaces
        end = start + max( s-r, v-u )
    else
        // GENERAL CASE
        i = [ (r+s)/2 ]
        LastRow( a, r, i-1, b, u, v, LLpref )
        LastRowReverse( a, i+1, s, b, u, v, LLsuff )

        pos = u-1
        type = GAP
        vmax = LLpref[u-1] + Gap_Penalty + LLsuff[u-1]
        for j = u to v do
            tempmax = LLpref[j-1] + match( a[i], b[j] ) + LLsuff[j]
            if tempmax > vmax then
                pos = j
                type = CHAR
                vmax = tempmax

            tempmax = LLpref[j] + Gap_Penalty + LLsuff[j]
            if tempmax > vmax then
                pos = j
                type = GAP
                vmax = tempmax

        if type == GAP then // Equation 2.7
            LinearSpace( a, r, i-1, b, u, pos, start, middle )
            align_a[middle] = a[i]
            align_b[middle] = '-'
            LinearSpace( a, i+1, s, b, pos+1, v, middle+1, end )
        else // type == CHAR, Equation 2.6
            LinearSpace( a, r, i-1, b, u, pos-1, start, middle )
            align_a[middle] = a[i]
            align_b[middle] = b[pos]
            LinearSpace( a, i+1, s, b, pos+1, v, middle+1, end )
```

Figure 2.4: Pseudo-code for *Hirschberg*'s Algorithm

any optimal path stretches from the lower right corner to the upper left corner of the d.p. matrix guarantees that the optimal path intersects the row i. Once pos is found, the original problem reduces to finding the optimal alignment for two pairs of smaller sequences: $a[1..i]$ and $b[1..pos]$, and $a[i+1..m]$ and $b[pos+1..n]$. These two new problems are solved recursively using the *LinearSpace* algorithm. The resulting optimal paths are concatenated to obtain an optimal path for the original problem. The following paragraphs explain in detail how the value of pos is determined for a given problem.

In an optimal alignment, $a[i]$ is aligned with either

1. a character $b[j]$, with $1 \leq j \leq n$, or

2. a gap that lays between $b[j]$ and $b[j+1]$, with $0 \leq j \leq n$.

In the second alternative, $j = 0$ accommodates the case when the gap is before the first character of b, while $j = n$ accommodates the case when the gap is after the last character.

Let *optimal* $\begin{pmatrix} x \\ y \end{pmatrix}$ denote an optimal alignment between the sequences x and y, and '+' denote concatenation of alignments. If *optimal* $\begin{pmatrix} a \\ b \end{pmatrix}$ satisfies alternative 1, then

$$optimal \begin{pmatrix} a \\ b \end{pmatrix} = optimal \begin{pmatrix} a[1..i-1] \\ b[1..j-1] \end{pmatrix} + \begin{matrix} a[i] \\ b[j] \end{matrix} + optimal \begin{pmatrix} a[i+1..m] \\ b[j+1..n] \end{pmatrix}. \tag{2.6}$$

If it satisfies alternative 2, then

$$optimal \begin{pmatrix} a \\ b \end{pmatrix} = optimal \begin{pmatrix} a[1..i-1] \\ b[1..j] \end{pmatrix} + \begin{matrix} a[i] \\ - \end{matrix} + optimal \begin{pmatrix} a[i+1..m] \\ b[j+1..n] \end{pmatrix}. \tag{2.7}$$

In both alternatives, the alignments of the subsequences are, in turn, optimal. Otherwise, an alignment strictly better than *optimal* $\begin{pmatrix} a \\ b \end{pmatrix}$ could be built.

Equations 2.6 and 2.7 show how to compute an optimal alignment recursively provided that, for any given i, it can be determined which of the two cases apply and what is the corresponding value of j. This problem can be solved by computing the optimal score between $a[1..i-1]$ and all the prefixes of b, and between $a[i+1..m]$ and all the suffixes of b. These scores correspond to the n alignments from the right hand side of Formula 2.6 ($1 \leq j \leq n$), and the $n+1$ alignments from the right hand side of Formula 2.7 ($0 \leq j \leq n$). From the $2n+1$ scores computed by Formula 2.6 and Formula 2.7, a maximal score is identified and the index j that generated this score is chosen as the splitting point for recursion.

The algorithm *LastRow* is used to compute the optimal scores between $a[1..i-1]$ and all the prefixes of b in linear space. A slightly modified version of the *LastRow* algorithm, called *LastRowReverse*, computes the optimal scores between $a[i+1..m]$ and all the suffixes of b. The algorithm *LinearSpace* (Figure 2.4) is invoked by the call

$$LinearSpace(a, 1, m, b, 1, n, 1, len),$$

which returns an optimal alignment in the global variables *align_a* and *align_b*. The length of this alignment is returned in *len*. *LinearSpace* can be easily modified to also return the score of the optimal alignment.

Hirschberg's algorithm saves space at the expense of recomputing some regions of the dynamic programming matrix. In fact, for the same input sequences, *Hirschberg*'s algorithm could compute as many as double the number of entries computed by *Needleman–Wunsch*'s algorithm. This theoretical result is proven in the remainder of this section.

Let $T(m, n)$ be the number of d.p. matrix entries computed following a call to

$$LinearSpace(a, r, s, b, u, v, start, end),$$

where $m = s - r + 1$ and $n = v - u + 1$. It should be noted that d.p. matrix entries are computed only inside *LastRow* and *LastRowReverse*. The computation of an entry consists of a *max* operation with three operands. Because *LinearSpace* spends most of its time computing d.p. matrix entries, it safe to assume that the total execution time of the algorithm is proportional to $T(m, n)$. Setubal and Meidanis [Setubal and Meidanis, 1997] give an upper bound for $T(m, n)$ by showing that

$$T(m, n) \leq 2mn.$$

The proof is based on mathematical induction on m. For $m = 1$, *LinearSpace* enters the base case section (i.e., BASE CASE in Figure 2.4), and no entry is computed. Obviously, $T(1, n) = 0 \leq 2n$ is true. For an $m > 1$, it is assumed by induction that the inequality holds for all values less than m. The induction step requires proving that the inequality is true for m. For the considered m, the call to *LastRow* incurs no more than $\frac{m}{2}n$ computations of d.p. entries. The same is true for the call to *LastRowReverse*. The two recursive calls to *LinearSpace* involve at most $T(\frac{m}{2}, j)$ and $T(\frac{m}{2}, n-j)$ computations of entries. After adding everything, and using the induction hypothesis, we conclude that

$$\begin{aligned} T(m, n) &\leq \tfrac{mn}{2} + \tfrac{mn}{2} + T(\tfrac{m}{2}, j) + T(\tfrac{m}{2}, n - j) \\ &\leq mn + mj + m(n - j) \\ &\leq 2mn, \end{aligned} \tag{2.8}$$

which confirms the claim [Setubal and Meidanis, 1997].

Finally, it should be noted that the above upper bound for $T(m, n)$ can be reached in a worst case scenario. It happens if, at every recursion level inside *LinearSpace*, the second sequence is also split in halves by the algorithm, i.e., always $j = \frac{n}{2}$.

2.3 The *Fast Linear Space Alignment* Algorithm

Needleman–Wunsch's algorithm finds an optimal alignment of two sequences without having to recompute any entry in the dynamic programming matrix. This is accomplished at the expense of using quadratic space, which is unacceptable for the alignment of long sequences. *Hirschberg*'s algorithm uses only linear space to find an optimal alignment. To achieve this goal, the algorithm must recompute some entries of the d.p. matrix. The total number of re-computations is close to, but not greater than, the total number of entries in the d.p. matrix for the two sequences.

The Fast Linear Space Alignment (*FastLSA*) algorithm [Charter *et al.*, 2000] builds on the observation that Hirschberg's approach of reducing the space requirements for the sequence alignment operation is too inflexible. In today's computers, more memory is available than the $2n$ d.p. matrix entries' worth of storage that are needed by *Hirschberg*'s algorithm in order to produce an optimal alignment for two sequences of lengths m and n. *FastLSA* takes advantage of the extra memory by caching the values of some entries of

```
Algorithm FastLSA
    input : logical-d.p.-matrix flsaProblem,
            cached-values cacheRow and cacheColumn,
            solution-path flsaPath
    output: optimal path corresponding to flsaProblem prepended to flsaPath

    /* Figure 2.6 (a) */
    if flsaProblem fits in allocated buffer then
        // BASE CASE
        /* Figure 2.6 (b) */
        return solveFullMatrix( flsaProblem, cacheRow, cacheColumn, flsaPath )

    // GENERAL CASE
    flsaGrid = allocateGrid( flsaProblem )
    initializeGrid( flsaGrid, cacheRow, cacheColumn )

    /* Figure 2.6 (c) */
    fillGridCache( flsaProblem, flsaGrid )

    newCacheRow = CachedRow( flsaGrid, flsaProblem.bottomRight )
    newCacheColumn = CachedColumn( flsaGrid, flsaProblem.bottomRight )

    /* Figure 2.6 (d) */
    flsaPathExt = FastLSA( flsaProblem.bottomRight, newCacheRow, newCacheColumn, flsaPath )

    while flsaPathExt not fully extended
        flsaSubProblem = UpLeft( flsaGrid, flsaPathExt )
        newCacheRow = CachedRow( flsaGrid, flsaSubProblem )
        newCacheColumn = CachedColumn( flsaGrid, flsaSubProblem )
        /* Figure 2.6 (e) */
        flsaPathExt = FastLSA( flsaSubProblem, newCacheRow, newCacheColumn, flsaPathExt )

    deallocateGrid( flsaGrid )

    /* Figure 2.6 (f) */
    return flsaPathExt
```

Figure 2.5: Pseudo-Code for *FastLSA*

the d.p. matrix, with the goal of reducing the number of re-computations that are needed to retrieve an optimal alignment. *FastLSA* still uses linear space, and its re-computation time is shown to be smaller than that of *Hirschberg*'s algorithm. Details of the *FastLSA* algorithm follow.

Suppose that $a[1..m]$ and $b[1..n]$ are the two biological sequences that must be aligned. Let RM denote the number of memory units (e.g., words) available for solving the sequence alignment problem. If $RM > m \times n$, then a full matrix algorithm (e.g., *Needleman–Wunsch*) can be used to solve the problem because the d.p. matrix can be stored in the available memory. Otherwise, *FastLSA* or *Hirschberg*'s algorithm can be used. The authors of *FastLSA* argue that their algorithm should be the choice [Charter *et al.*, 2000] .

FastLSA is a recursive algorithm based on the *divide and conquer* paradigm. A call to *FastLSA* takes as input a logical d.p. matrix corresponding to a pair of sequences and an optimal solution path that ends at the bottom-right entry of this logical d.p. matrix. *FastLSA* prepends to the input path an optimal path which traverses the input matrix from the bottom-right entry to the top or the left boundary. The resulting optimal path constitutes the output of *FastLSA*. A row and a column of cached d.p. matrix entry values are also passed in with each call to *FastLSA*. The pseudo-code for the *FastLSA* algorithm is shown in Figure 2.5.

For the initial call to *FastLSA*, the logical d.p. matrix used as input – *flsaInitialProblem* – corresponds to the input sequences a and b. The attribute "logical" is used because only the shape of the matrix is known initially. This initial logical d.p. matrix has $(m+1) \times (n+1)$ entries whose values must be computed. The values of the initial cache row, *cacheRow*, and cache column, *cacheColumn*, are computed as in Equation 2.1 and Equation 2.2, respectively. The initial optimal path, *flsaInitialPath*, is formed from a single point, (m, n), the bottom-right entry of the original logical d.p. matrix. *FastLSA* is invoked by the call

$$solPath = FastLSA(flsaInitialProblem, cacheRow, cacheColumn, flsaInitialPath)$$

which will return a partial optimal path in *solPath*. This partial optimal path can then be extended to the top-left entry of the logical d.p. matrix to form a complete optimal path.

Prior to running *FastLSA*, BM units of memory are reserved from the RM units available. These reserved units are subsequently referred to as the *Base Case buffer*. If the d.p. matrix corresponding to the input problem can be allocated in the Base Case buffer, then an optimal path for the input problem is built using a full matrix algorithm. This corresponds to the BASE CASE section of the algorithm (lines 1–2 in Figure 2.5).

The full matrix algorithm uses the input values *cacheRow* and *cacheColumn* as the first row and column of the d.p. matrix it must compute (Figure 2.6 (a)). After all entries of the d.p. matrix have been computed, an optimal path through the matrix is built. Figure 2.6 (b) shows the computed and stored d.p. matrix entries of a sample base case. In this figure, an optimal path is found to extend from the bottom-right corner entry, A, to the top boundary entry, B.

If the size of the d.p. matrix for the input problem is bigger than BM, the GENERAL CASE of the algorithm is followed (Figure 2.5). In this case, *FastLSA* splits the input problem into smaller subproblems. These subproblems are solved recursively using calls to *FastLSA*. The solution paths for these subproblems, if concatenated, form a solution path for the input problem.

The general case of *FastLSA* starts by dividing each dimension of the logical d.p. matrix into k equal segments, $k \geq 2$. As a result, the d.p. matrix for the input problem is partitioned into k^2 *logical sub-matrices* of size approximately $\frac{m}{k} \times \frac{n}{k}$ (Figure 2.6 (c)). These sub-matrices are laid out in k rows, each row having k columns.

The first goal of the general case is to find the values of the entries of the d.p. matrix which lie on the left and upper border of the k^2 logical sub-matrices, and save them. These interesting values lie exactly along k rows and k columns of the logical d.p. matrix. The grid *flsaGrid* is allocated in order to store these values once they are computed (line 3 of

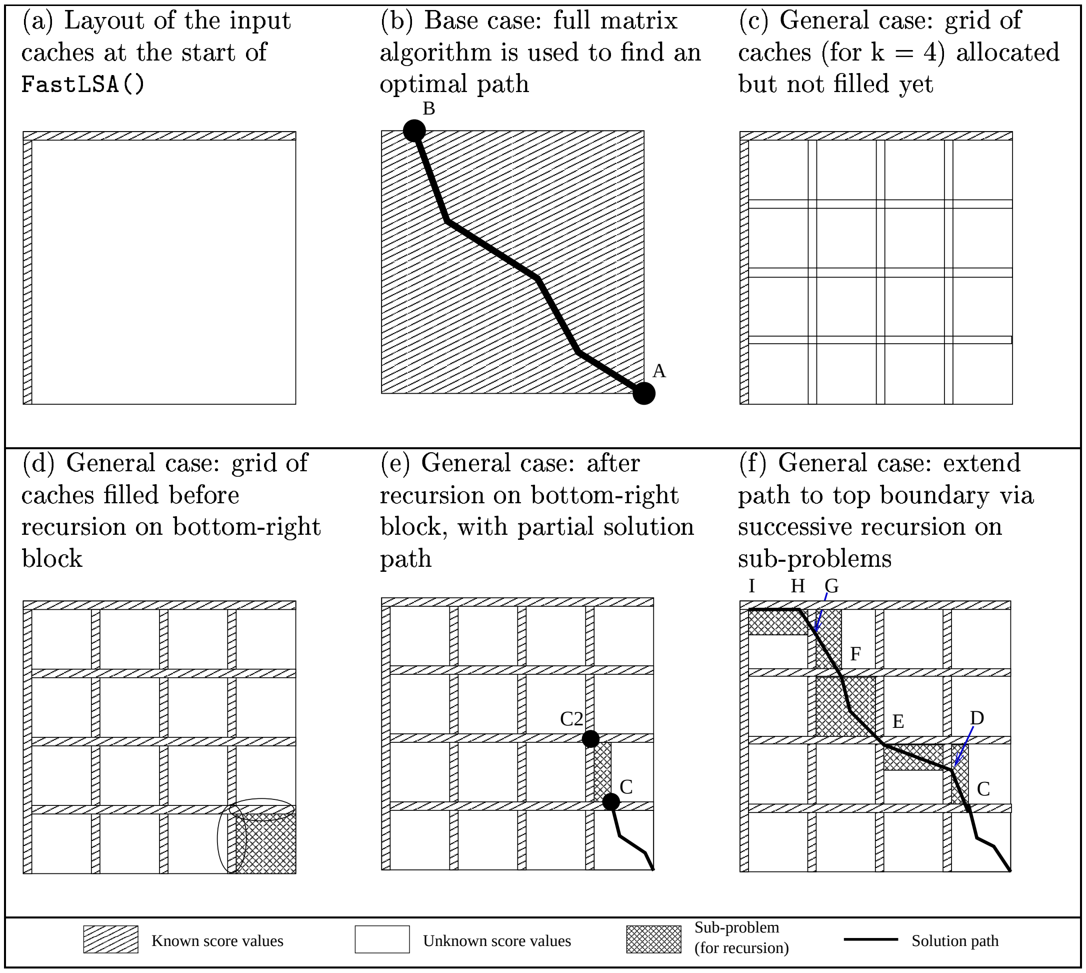

(a) Layout of the input caches at the start of FastLSA()
(b) Base case: full matrix algorithm is used to find an optimal path
B
A
(c) General case: grid of caches (for k = 4) allocated but not filled yet
(d) General case: grid of caches filled before recursion on bottom-right block
(e) General case: after recursion on bottom-right block, with partial solution path
C2
C
(f) General case: extend path to top boundary via successive recursion on sub-problems
I
H
G
F
E
D
C
Known score values
Unknown score values
Sub-problem (for recursion)
Solution path

Figure 2.6: Execution Stages of *FastLSA*

the pseudo-code). *flsaGrid* consists of k rows of size n and k columns of size m. The grid rows and columns can be seen as overlapping the rows and column of the d.p. matrix.

The uppermost row and the leftmost column of *flsaGrid* will hold the values passed in with the row *cacheRow* and the column *cacheColumn*. This initialization of the grid is done in *initializeGrid* (line 4 in Figure 2.5). Figure 2.6 (c) shows this stage of the computation.

In order to fill the remaining $k-1$ rows and $k-1$ columns of the grid *flsaGrid*, all the entries of the d.p. matrix are computed, except for those forming the bottom-right sub-matrix. This is accomplished with the call to *fillGridCache* (line 6 in Figure 2.5). The algorithm that is used to compute the entries of the d.p. matrix is similar to the algorithm *LastRow* from Figure 2.3, Section 2.2. The only difference is that the values of the entries lying on the right-most column and the bottom-most row of a sub-matrix are stored in *flsaGrid*, as soon as they are computed. These entries are saved in the portions of *flsaGrid* that they overlap. However, there are two exceptions: only the right-most column is saved from the sub-matrices of the kth row of sub-matrices, and only the bottom-most row is saved from the sub-matrices of the kth column of sub-matrices. The entries corresponding to the bottom-right sub-matrix are not yet computed.

Figure 2.6 (d) shows *flsaGrid* completely filled before the *FastLSA* is applied recursively to the bottom-right sub-matrix. The portions from *flsaGrid* that border the bottom-right sub-matrix are passed with the recursive call to *FastLSA* as the new caches *newCacheRow* and *newCacheColumn* (line 8 of the pseudo-code). When this recursive call to *FastLSA* returns, the optimal path for the initial problem has been extended from the bottom-right entry to the entry C (Figure 2.6 (e)). Note that C could also have been on the left boundary of the bottom-right sub-matrix.

The next step of the general case is to extend the optimal path from the entry C to an entry on the left or upper boundary of the initial logical d.p. matrix. This step is accomplished through successive recursive calls to *FastLSA* in the `while`-loop of lines 9–13 in Figure 2.5.

Note that during this latter step, calls to *FastLSA* are not necessarily applied to entire sub-matrices. Every time the optimal path extends into a new sub-matrix, the next sub-problem to be solved by *FastLSA* is identified through a call to *UpLeft* (line 10 in Figure 2.5). The coordinates of this new logical d.p. matrix are computed by *UpLeft* as follows:

- the top-left corner of the new logical matrix is given by the top-left corner of the sub-matrix that is to be entered next by the optimal path;
- the bottom-right corner of the new logical matrix is given by the head of the current optimal path.

Figure 2.6 (e) shows the logical d.p. matrix found by *UpLeft* when first called in the `while`-loop. The top-left corner of the new logical matrix is *C2*, with C, the head of the current optimal path, being the bottom-right corner. Then, the portions from *flsaGrid* which border this new logical d.p. matrix to North and West are identified. These are the new caches which are passed with the recursive call to *FastLSA* as *newCacheRow* and *new-CacheColumn*. When this recursive call returns, the optimal path for the original problem has been extended from C to the entry D (Figure 2.6 (f)). At the end of the second cycle of the `while`-loop, the optimal path has been further extended to the entry *E*.

In the remaining cycles of the `while`-loop, the optimal path is further extended through the sub-matrices of the input matrix until the head of the path intersects the first row or the first column of the grid. Figure 2.6 (f) shows the optimal path being extended through entries *E*, *F*, *G*, and *H*. The `while`-loop stops when H becomes the head of the current optimal path because H lies on the first row of *flsaGrid*. Next, the grid of caches *flsaGrid* is deallocated, and the initial call to *FastLSA* returns. The optimal path corresponding to the input logical d.p. matrix is returned to the initial caller. The returned path extends from the bottom-right corner of the original input matrix to the entry *H*.

After the initial invocation of *FastLSA* returns, the partial optimal path *solPath* is further extended to the top-left corner along the first row or the first column of the d.p. matrix.

Figure 2.6 (f) shows the partial optimal path being extended to the top-left corner I along the first row of the d.p. matrix. The resulting optimal path corresponds uniquely to an optimal alignment between the input sequences a and b. This correspondence is introduced in Section 2.1, page 10.

It is useful to observe that *FastLSA* solves a succession of rectangular problems, called *FastLSA subproblems*, using either a Base Case approach for the small subproblems, or a Fill Cache approach for the subproblems that do not fit in the Base Case buffer. The subproblems solved as Base Cases are referred to as *Base Case subproblems*. The subproblems solved in the General Case are referred to as *Fill Cache subproblems*.

FastLSA uses more space than *Hirschberg*'s algorithm. This gives *FastLSA* the advantage of recomputing fewer entries in the d.p. matrix, thus improving the time performance of the sequence alignment operation. The space required by *FastLSA* is still linear in the size of the input sequences as will be shown next, based on the results presented in [Charter *et al.*, 2000] and [Charter *et al.*, 2002]. Furthermore, *FastLSA* can be adjusted to use all RM units of memory that are available.

Let $S(m,n,k)$ be the maximum number of d.p. matrix entries that need to be stored in order to align the sequences $a[1..m]$ and $b[1..n]$, using a grid cache of k rows and k columns. If the General Case of the algorithm is followed for the initial call to *FastLSA*, $k-1$ rows of length n and $k-1$ columns of length m must be allocated for the grid cache. The initial cache row and cache column which are passed as arguments to the *FastLSA* call are used as the top-most row and the left-most column of the grid. They have already been allocated by the caller function, and this is why they are not counted as part of $S(m,n,k)$. The cache in the first call to *FastLSA* uses $(k-1) \times (m+n)$ entries in total.

The recursive call to the bottom-right sub-problem uses at most $S(\frac{m}{k}, \frac{n}{k}, k)$ space. Because all the subproblems solved inside the `while`-loop are equal to or smaller than the bottom-right sub-problem, $S(\frac{m}{k}, \frac{n}{k}, k)$ is a good upper bound for the space used by the recursive calls to *FastLSA* generated by the initial call. After putting everything together, we get

$$S(m,n,k) = (k-1) \times (m+n) + S(\tfrac{m}{k}, \tfrac{n}{k}, k) \tag{2.9}$$

The recursive relation for space becomes

$$\begin{aligned} S(m,n,k) &= (k-1) \times (m+n) + (k-1) \times (\tfrac{m}{k} + \tfrac{n}{k}) + S(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k) \\ &= (k-1) \times (m+n) \times (1 + \tfrac{1}{k}) + S(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k) \\ &= \cdots \\ &= (k-1) \times (m+n) \times (1 + \tfrac{1}{k} + \cdots + \tfrac{1}{k^{a-1}}) + S(\tfrac{m}{k^a}, \tfrac{n}{k^a}, k). \end{aligned} \tag{2.10}$$

Because the space for Base Case subproblems is allocated in the Base Case buffer, it is true that $S(\frac{m}{k^a}, \frac{n}{k^a}, k) \leq BM$, and Equation 2.10 becomes

$$\begin{aligned} S(m,n,k) &\leq (k-1) \times (m+n) \times \frac{1-\frac{1}{k^a}}{1-\frac{1}{k}} + BM \\ &= k \times (m+n) \times (1 - \tfrac{1}{k^a}) + BM \\ &\leq k \times (m+n) + BM. \end{aligned} \tag{2.11}$$

Equation 2.11 shows that *FastLSA* uses linear space. It also provides the means to compute k and BM when the space utilization is to be maximized.

With regard to the time complexity, let $T(m,n,k)$ be the number of d.p. matrix entries computed by *FastLSA* when the sequences a and b are aligned using a grid cache with k rows and k columns. It can be proven ([Charter *et al.*, 2000], [Charter *et al.*, 2002]) that, in the worst case scenario,

$$T(m,n,k) = m \times n \times \tfrac{k+1}{k-1}. \tag{2.12}$$

It should be noted that the total execution time of *FastLSA* is proportional to $T(m,n,k)$.

As mentioned throughout this section, *FastLSA* trades space for performance. For example, when $k=5$, $T(m,n,5) = 1.5 \times m \times n$. This is theoretically a lower upper bound for time complexity than the upper bound obtained for *Hirschberg*'s algorithm (Equation 2.8). The upper bound provided by *FastLSA* decreases when the value of k increases.

2.4 Other Pairwise Sequence Alignment Algorithms

In the previous sections, we focus on algorithms that produce an optimal alignment for a pair of sequences. We describe these algorithms in detail because *Parallel FastLSA* is directly related to *FastLSA* and draws on the techniques used in *Hirschberg*'s algorithm and *Needleman–Wunsch*'s algorithm.

All three algorithms presented so far compute the entries of a d.p. matrix in order to find an optimal solution to the alignment problem. However, there are algorithms that use A^*-type search to find an optimal alignment between two sequences. For example, *DCFS* (Divide-and-Conquer Frontier Search) [Korf and Zhang, 2000] finds an optimal path from the upper-left corner of the d.p. matrix to the bottom-right corner using A^* unidirectional search. This algorithm is "more efficient in both time and space than the bidirectional version", *DCBDS* (Divide & Conquer Bidirectional Search) [Korf, 1999].

Other algorithms, such as *BLAST* (Basic Local Alignment Search Tool) [Altschul *et al.*, 1990], sacrifice the optimality of the results for improved performance. *BLAST* is a rapid heuristic algorithm which produces biologically significant alignments through direct approximation of alignments that "optimize a measure of local similarity, the maximal segment pair (MSP) score" [Altschul *et al.*, 1990]. Based on the analysis presented by Altschul *et al.*, *BLAST* was deemed to be "an order of magnitude faster than existing sequence comparison tools of comparable sensitivity" [Altschul *et al.*, 1990]. *BLAST* is

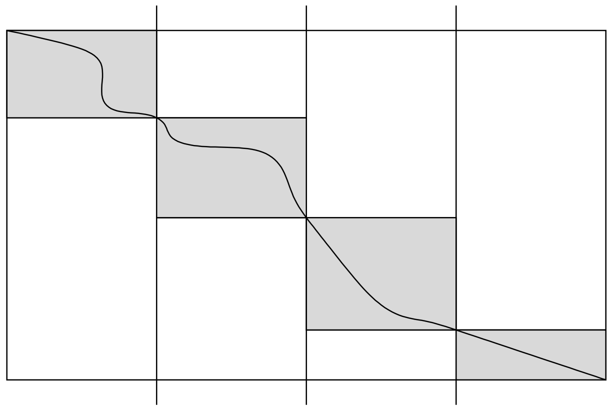


Figure 2.7: *Parallel Space-Saving* Algorithm

primarily used for finding biological similarities between a query sequence and a *database*, which is a large collection of sequences.

The last generation of *BLAST* programs includes two popular tools: *Gapped BLAST* and *PSI-BLAST* [Altschul *et al.*, 1997]. *Gapped BLAST* is an improved version of *BLAST* that uses a heuristic for generating gapped alignments. *PSI-BLAST* (Position-Specific Iterated BLAST) automatically combines the alignments that are produced by *BLAST*, and are statistically significant, into a position-specific score matrix. It then searches the database using this matrix. *PSI-BLAST* is deemed to be much more sensitive to weak, but biologically relevant, similarities than *Gapped BLAST*. Because of the added features, these algorithms are approximately three times slower than the original *BLAST*.

To our knowledge, there is not a great deal of literature on parallel algorithms for pairwise biological sequence alignment. Aluru *et al.* [Aluru *et al.*, 1999] suggest an embarrassingly parallel algorithm for sequence alignment, which they refer to as the *Parallel Space-Saving* algorithm. Embarrassingly parallel algorithms achieve good speedups because the processors are kept busy with useful computation, and little synchronization is required.

The *Parallel Space-Saving* algorithm is a straight-forward generalization of *Hirschberg*'s algorithm. One of the sequences to be aligned is divided into p sub-sequences of equal length. The idea is to find p sub-sequences of the other sequence so that the total alignment is obtained by aligning the p pairs of smaller sub-sequences and concatenating their solutions. To find the corresponding pairs, all the entries in the d.p. matrix for the two sequences are computed using a linear space algorithm. During the computation of these entries, the intersections of an optimal path with p equidistant cache columns are identified (Figure 2.7). The original problem is then split into p subproblems that will be solved sequentially on each of the p processors. For $p = 2$, the original *Hirschberg*'s algorithm is obtained.

The drawback of this parallel algorithm is the lack of control on the granularity of the p subproblems it generates. To achieve good speedups, the p subproblems, depicted as grayed rectangles in Figure 2.7, should have similar sizes. Because one sequence is divided in p

equally sized sub-sequences, the corresponding sub-sequences in the other sequence should be equally sized as well. This is unlikely to happen in practice because of the irregular nature of the biological sequences to be aligned. Our analysis indicates that, while of theoretical interest, the algorithm suggested in Section 5 of [Aluru *et al.*, 1999] may not be useful in practice.

A parallel version of the *Needleman–Wunsch* algorithm is introduced in [Martins *et al.*, 2001]. The d.p. matrix is divided into equally sized blocks, and the algorithm statically preassigns rows of blocks to each processor. This algorithm suffers from the same major drawback as the original *Needleman–Wunsch* algorithm: the space required is quadratic in the size of the sequences. The particular implementation considered in [Martins *et al.*, 2001] is based on EARTH, "a fine-grain event-driven multi-threaded execution and architecture model" [Martins *et al.*, 2001]. The performance numbers presented, although impressive, are obtained through simulation, and the largest d.p. matrix computed for their benchmarks has only $4,000 \times 10,000$ entries.

2.5 Concluding Remarks

In this chapter we present three algorithms that are closely related to *Parallel FastLSA*. In fact, *FastLSA* is the sequential version of *Parallel FastLSA*, and constitutes the starting point in the development of *Parallel FastLSA*. The algorithms of *Needleman–Wunsch* and *Hirschberg* are considered classic algorithms for optimal pairwise sequence alignment, and are the most widely used algorithms among those based on dynamic programming.

In summary, *Needleman–Wunsch*'s algorithm computes and stores all the entries of the d.p. matrix; therefore, it has quadratic time and space complexity. *Hirschberg*'s algorithm uses only linear space, but this dramatic reduction in space comes at the expense of its time complexity: in the worst case scenario, the total number of d.p. matrix entries computed by *Hirschberg*'s algorithm is double that of the d.p. matrix entries computed by *Needleman–Wunsch*'s algorithm. *Hirschberg*'s algorithm must recompute the value of some entries because it uses only as much memory as needed to store one of the sequences, even if more memory is available.

FastLSA improves on *Hirschberg*'s algorithm by taking advantage of the extra memory available and storing some of the computed d.p. matrix entries in a Grid Cache. The benefit of having a Grid Cache is that a smaller number of entries are recomputed than in the case of *Hirschberg*'s algorithm. Consequently, *FastLSA* is faster than *Hirschberg*'s algorithm, both theoretically and empirically [Charter *et al.*, 2002].

Chapter 3

Parallel FastLSA

We performed extensive experiments with the three algorithms discussed in the previous chapter, and concluded that *FastLSA* is the most effective when used to align very long sequences. *FastLSA* outperforms its closest competitor, *Hirschberg*'s algorithm, when DNA sequences as short as 10 Kbp are aligned [Charter *et al.*, 2002]. However, the theoretical time of *FastLSA* still has quadratic complexity and the real turnaround time increases dramatically with the increase in size of the sequences. In order to alleviate this problem, we have developed a parallel version of the *FastLSA* algorithm, subsequently referred to as the *Parallel FastLSA* algorithm. The main contribution of the thesis is the design, implementation, and evaluation of *Parallel FastLSA*.

3.1 Description of the *Parallel FastLSA* Algorithm

Parallel FastLSA improves the execution time of the original *FastLSA* algorithm by parallelizing its two major time-consuming components:

1. Base Case: the full matrix algorithm used for solving Base Case subproblems (line 2 of the pseudo-code from Figure 2.5), and

2. General Case: the computation of the FastLSA Grid Cache for the Fill Cache subproblems (line 5 of the pseudo-code from Figure 2.5).

The pseudo-code for *Parallel FastLSA* is shown in Figure 3.1. The only changes from the sequential version are the replacement of the sequential *solveFullMatrix*() with a parallel version, *parallelSolveFullMatrix*(), in line 2, and the replacement of the sequential *fillGridCache*() with a parallel version, *parallelFillGridCache*(), in line 5. No other component of the algorithm is executed concurrently.

In our experiments with *Parallel FastLSA*, we discovered that parallelism benefits only the Fill Cache subproblems. In all the experiments we performed with our choice of parameter values, the Base Case subproblems took longer to solve in parallel than sequentially. For this reason, in the following chapter we analyze the performance of an implementation

```
Algorithm Parallel FastLSA
    input : logical-d.p.-matrix flsaProblem,
            cached-values cacheRow and cacheColumn,
            solution-path flsaPath
    output: optimal path corresponding to flsaProblem prepended to flsaPath

    /* Figure 2.6 (a) */
1   if flsaProblem fits in allocated buffer then
        // BASE CASE
        /* Figure 2.6 (b) */
2       return parallelSolveFullMatrix( flsaProblem, cacheRow, cacheColumn, flsaPath )

    // GENERAL CASE
3   flsaGrid = allocateGrid( flsaProblem )
4   initializeGrid( flsaGrid, cacheRow, cacheColumn )

    /* Figure 2.6 (c) */
5   parallelFillGridCache( flsaProblem, flsaGrid )

6   newCacheRow = CachedRow( flsaGrid, flsaProblem.bottomRight )
7   newCacheColumn = CachedColumn( flsaGrid, flsaProblem.bottomRight )

    /* Figure 2.6 (d) */
8   flsaPathExt = ParallelFastLSA( flsaProblem.bottomRight, newCacheRow, newCacheColumn, flsaPath )

9   while flsaPathExt not fully extended
10      flsaSubProblem = UpLeft( flsaGrid, flsaPathExt )
11      newCacheRow = CachedRow( flsaGrid, flsaSubProblem )
12      newCacheColumn = CachedColumn( flsaGrid, flsaSubProblem )
        /* Figure 2.6 (e) */
13      flsaPathExt = ParallelFastLSA( flsaSubProblem, newCacheRow, newCacheColumn, flsaPathExt )

14  deallocateGrid( flsaGrid )

    /* Figure 2.6 (f) */
15  return flsaPathExt
```

Figure 3.1: Pseudo-Code for *Parallel FastLSA*

of *Parallel FastLSA* that solves all Base Case subproblems sequentially. However, we still explain how the Base Case subproblems can be solved in parallel, because a different choice of parameter values can potentially make their implementation in parallel efficient. In the remainder of this section, we describe how the parallel work is organized, first for the Base Case subproblems, and then for the Fill Cache subproblems.

As explained in Section 2.3, *FastLSA* stops recursing when the input logical d.p. matrix *flsaProblem* can be allocated in the Base Case buffer (line 1 in Figure 3.1). The optimal path corresponding to this matrix is determined using a full matrix algorithm (e.g., *Needleman–Wunsch*). For the parallel version of the full matrix algorithm, the dynamic programming matrix is allocated in shared memory. As in the sequential version of *FastLSA*, the initial values for the d.p. matrix are provided by the calling function. They are passed in as the cache row *cacheRow* and the cache column *cacheColumn*. These initial values are also stored in shared memory, and they are essential for starting the computation of the d.p. matrix. In order to compute the value of a d.p. matrix entry, the values of the adjacent entries from

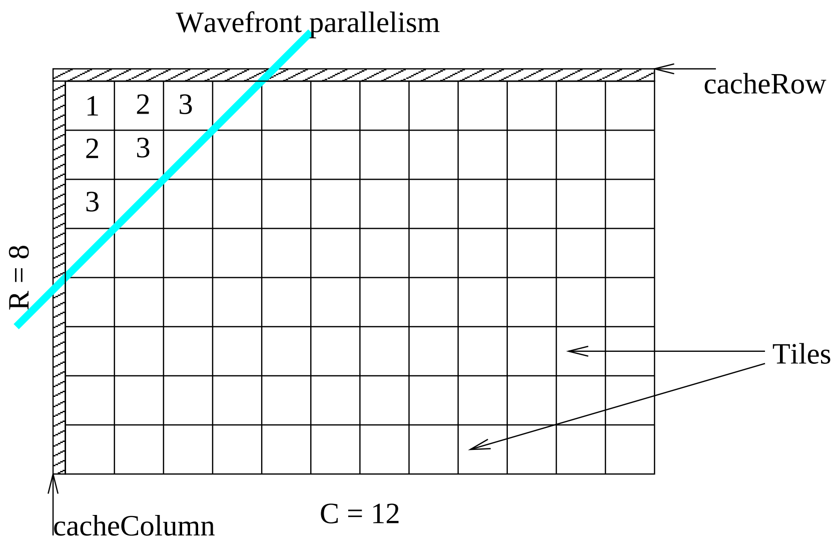


Figure 3.2: Data Partitioning for Parallel Base Case Subproblems

North, West, and North–West must be available as required by Formula 2.3.

The d.p. matrix is logically partitioned in $R \times C$ equally sized rectangular regions, with $R \geq 1$ and $C \geq 1$. Note that in Figure 3.2, $R = 8$ and $C = 12$ are just examples. These regions, subsequently referred to as *tiles*, are laid out along R rows, each row having C columns. At any moment during the parallel processing of the d.p. matrix, a processor is either idle, or it is working on only one tile. Furthermore, only one processor can work on a tile. Once the processing of a tile ends, no processor will work on that tile again.

The parallel processing starts with one processor computing the entries of the top-left tile, using a Full Matrix algorithm. The top-left tile is labelled 1 in Figure 3.2. The computation of the top-left tile is possible because the initial row and column values for this tile are available. In fact, the top-left tile is the only tile that has all its initial values available. These initial values come from the entries of *cacheRow* and *cacheColumn* which border the top-left tile. All the other processors are idle during this first step. After the top-left tile is processed, all the values of its corresponding entries can be found in shared memory.

After the first step, there is enough information available to start computing the entries in the tiles which neighbor the top-left tile to East and South. For example, for the tile placed East from the top-left tile (i.e., in row 1 and column 2 of the array of tiles), the initial row values come from the entries of *cacheRow* that border the tile, while the initial column values come from the entries of the right-most column of the top-left tile. The two tiles neighboring the top-left tile can be computed in parallel on two different processors.

The processing of the tiles advances on a diagonal-like front. In Figure 3.2, each diagonal of tiles labeled with the same number forms a *wavefront line.* At the P^{th} step, all the P processors can work in parallel because the wavefront line consists of exactly P tiles. The

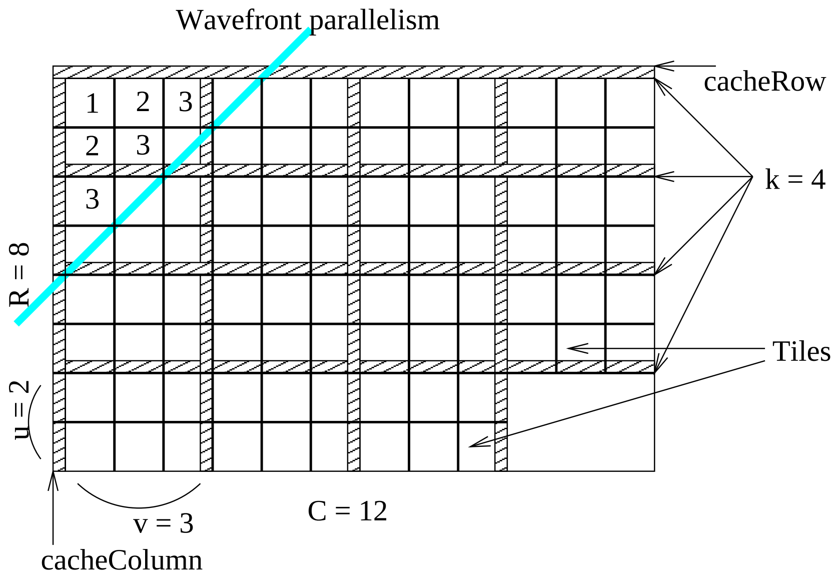


Figure 3.3: Data Partitioning for Parallel Fill Cache Subproblems

parallel computation ends when all the $R \times C$ tiles have been computed. More details on how the parallel work is organized are provided in the next section.

When the parallel phase ends, all the d.p. matrix entries are available in shared memory. As in the sequential version of the full matrix algorithm, one of the processors builds an optimal path which extends from the bottom-right corner of the d.p. matrix to its left or upper boundary.

For each Fill Cache subproblem, the logical dynamic programming matrix is already split in k^2 smaller matrices, the logical sub-matrices introduced in Section 2.3. However, the Fill Cache subproblems are much bigger than the Base Case subproblems and, in order to control the granularity of the parallel work, each of the $k^2 - 1$ sub-matrices that need to be computed in this phase is further divided into $u \times v$ equally sized tiles (Figure 3.3). The result is a grid of finer granularity than the FastLSA grid. This new grid partitions the d.p. matrix in $(k^2 - 1) \times u \times v$ tiles that are to be processed in parallel. These tiles are placed along $R = k \times u$ rows and $C = k \times v$ columns. In Figure 3.3, $k = 4$, $u = 2$, and $v = 3$ are examples of possible values for these parameters. Because of this choice of parameter values, the tiles are laid out as an array of $R = 8$ rows and $C = 12$ columns.

The parallel processing starts with one processor computing the entries of the top-left tile. The algorithm used to compute the entries corresponding to a tile is similar to the algorithm *LastRow* from Figure 2.3, Section 2.2. This algorithm computes the entries of the tile using linear space. The values of the entries forming the right-most column and the bottom-most row of the tile are saved in a special cache, referred to as *Tile Cache* (Figure 3.4). The Tile Cache and the Grid Cache are both allocated in shared memory.

The Tile Cache is needed in order to allow the parallel computation to progress. For example, after the right-most column and the bottom-most row of the top-left tile are saved

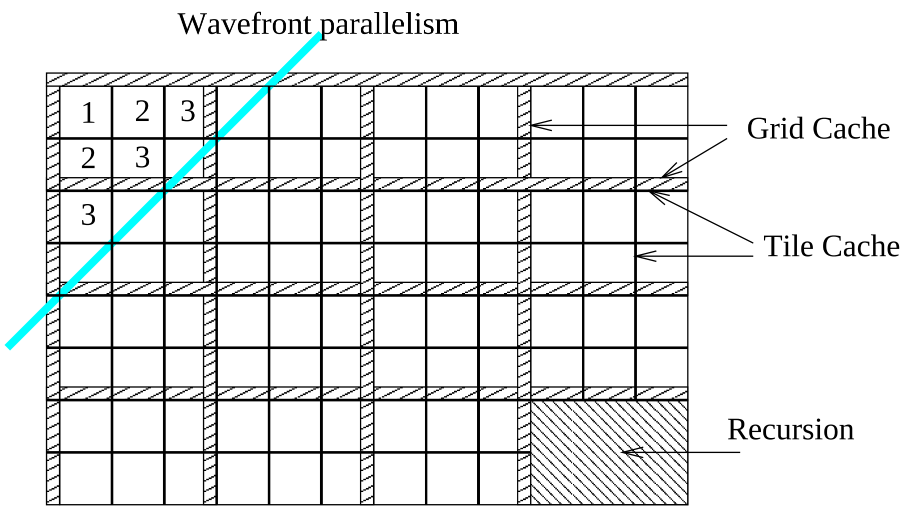


Figure 3.4: FastLSA Grid Cache and Tile Cache for Parallel Fill Cache Subproblems

in the Tile Cache, step 2 of the parallel processing can start. At step 2, two processors can start processing in parallel the two tiles which neighbor the top-left tile (i.e., the tiles labeled with 2 in Figure 3.4). For each of the two tiles, the initial row values and the initial column values are available from the Tile Cache. At the P^{th} step, all the P processors can work in parallel because the wavefront line consists of exactly P tiles. The parallel computation ends when all the $(k^2 - 1) \times u \times v$ tiles have been computed. More details on how the parallel work is organized are provided in the next section.

Figure 3.4 shows the Grid Cache delimiting the FastLSA sub-matrices and the Tile Cache delimiting the tiles. The bottom-right sub-matrix is not partitioned into tiles in this phase because it will be solved through a recursive call to *Parallel FastLSA*.

As can be seen in Figure 3.4, the Grid Cache always overlaps a subset of the Tile Cache, except for the boundaries of the bottom-right sub-matrix. The left-most column and the upper-most row of the two caches are initialized using the cache values received as input in *cacheColumn* and *cacheRow*, respectively. As mentioned above, the processor that computes the entries corresponding to a tile saves the entries from the right-most column and bottom-most row in the Tile Cache. These entries are also saved in the Grid Cache if they are overlapped by a Grid Cache column or a Grid Cache row. Note that the tiles in the bottom-most row (i.e., the R^{th} row) and those in the right-most column (i.e., the C^{th} column) form degenerate cases where only the right-most column or the bottom-most row is saved.

After all the tiles have been processed, the FastLSA Grid Cache has been filled and the Tile Cache can be deallocated. Then, *Parallel FastLSA* is applied recursively to the bottom-right sub-matrix (Figure 3.4). Note that new caches of each type, FastLSA Grid Cache and Tile Cache, are allocated in shared memory for each Fill Cache subproblem solved.

3.2 Implementation Details

As mentioned in the previous section, tiles cannot be processed in an arbitrary order. A tile can be processed only if the entries of the row preceding its top-most row, and the entries of the column preceding its left-most column are already in the Tile Cache. This means that the tile directly above a tile X, and the one immediately to the left of X, must have already been processed before X can be processed. This strict dependency is present for both the parallel full matrix algorithm and the parallel computation of the FastLSA Grid Cache. For this reason, the two types of parallel regions used by *Parallel FastLSA* can be implemented using the same strategy for the distribution of parallel tasks.

We have investigated two solutions to the problem of assigning the tiles that are ready to be processed to the processors that are available. In the first solution, the tiles that are ready to be processed are placed in a work queue, and a processor that needs work dynamically dequeues a tile from the queue. In the second solution, entire rows of tiles are preassigned to the processors, and each tile is processed as soon as it becomes ready. These two approaches are explained in detail in the following subsections.

3.2.1 Dynamic Distribution of Work

Initially, only the top-left tile, which is labelled 1 in Figure 3.4, can be processed because it is the only tile for which both the initial row and the initial column values are known. The top-left tile is placed in the work queue, which is allocated in shared memory. Every time parallel computation is performed, this queue contains references to the tiles that are ready to be processed. Inside *parallelFillGridCache*() and *parallelSolveFullMatrix*(), all processors try to grab a tile from the work queue and execute the task associated with it. For a Fill Cache subproblem, the task is to fill the cache entries adjacent to the tile and not known previously. For a Base Case subproblem, the task is to compute the values of the tile entries.

A processor that finds the queue empty is blocked until a tile becomes available for that processor. A reservation mechanism is used in order to avoid the starvation of certain processors, and to reduce the contention for the queue access. In essence, a monitor is associated to each queue slot.

After finishing working on its assigned tile, a processor checks to see if it can place in the queue the adjacent tile to the right, or the adjacent tile below. For example, a tile X, neighboring the current tile to the right, can be placed in the work queue if and only if the tile above X has also been processed. This condition ensures that both the initial row and the initial column values are known for X.

The condition stated above can be implemented by associating a counter to each tile. The counter of a tile X is incremented by the processor which processed the tile above or to the left of X. The processor which increments the value of the counter to 2 is also responsible for placing X in the work queue. Note that the tiles from the first row and the first column have their counters set to 1 initially, because some of the initial values for

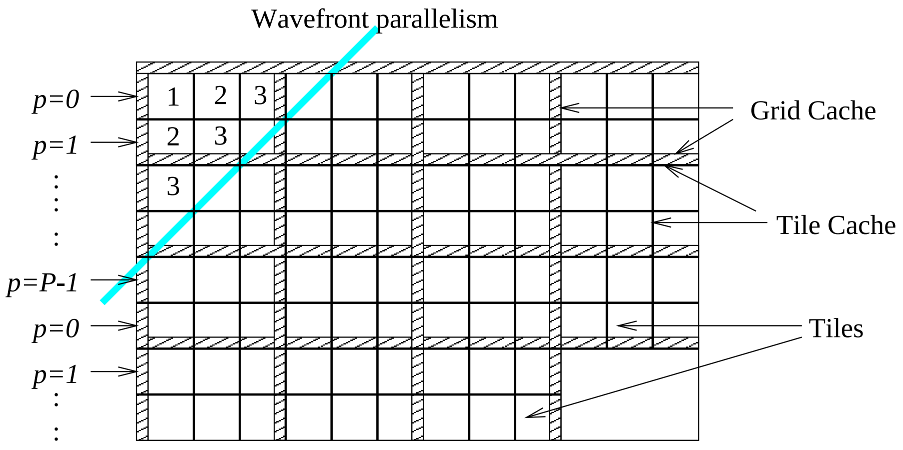


Figure 3.5: *Parallel FastLSA*: Static Distribution of Work

these tiles are already available. The counter of the top-left tile is set to 2 initially, while the counters of all the other tiles start at 0.

After the tile labelled 1 is computed, the tiles labelled 2 in Figure 3.4 can be placed in the work queue. After those tiles have been processed, more tiles (labelled 3) can be placed in the queue, in a pattern known as *wavefront parallelism.* Note that the tiles labelled 3 need not be placed in the work queue all at the same time. They become available for processing as soon as the tiles labelled 2 have been computed.

The parallel processing region ends when all the designated tiles have been computed. Filling the Grid Cache in parallel requires the processing of

$$(k^2 - 1) \times u \times v = R \times C - u \times v \text{ tiles,}$$

while the parallel full matrix algorithm computes the values of $R \times C$ tiles. Note that the values of R and C need not be the same for both the full matrix and cache filling computations. Furthermore, some of the tiles can be empty when R and C are bigger than the dimensions of the input d.p. matrix.

3.2.2 Static Distribution of Work

This is another solution to the problem of allocating the tiles, which are dependent in a wavefront manner, to the P processors available. As shown in Figure 3.5, each of the R rows of tiles is assigned to a processor in a circular fashion. The first processor (i.e., $p = 0$) starts by solving the top-left tile, which is the only one with initial row values and initial column values available. After the top left tile has been solved, the tiles labelled 2 in Figure 3.5 can also be computed. The first processor computes the second tile in the first row, while the second processor computes the first tile in the second row. As soon as the second processor finishes its first tile, the third processor can start working on its first tile, and so on.

The solution described above is a round-robin mechanism for work distribution, similar to that presented in [Martins *et al.*, 2001]. The static distribution of work solution deals with the dependency between tiles without using a queue or system locks. Each processor p busy waits until the tile above its current tile is solved by the processor $p-1 \pmod{P}$. At this point, p can start working on its current tile. When p finishes the last tile on its current row, r, it moves to the next row that was preassigned to it, $r+P$. If $r+P > R$, the row r is the last row on which the processor p worked. The entire computation finishes when all the tiles have been processed.

The busy waiting mechanism relies heavily on the capacity of each processor to refresh its cache quickly. Each time the processor p finishes solving a tile, an index is incremented, and the processor $p+1 \pmod{P}$ must be notified of the new value of the index. This is why each processor not working on a tile, continuously probes the index associated with the previous row.

3.3 Space and Time Complexity

We argue that *Parallel FastLSA* still uses linear space and that the time complexity of the algorithm is still quadratic. We prove this claim by finding a linear upper bound for the space complexity of *Parallel FastLSA* and by finding a quadratic upper bound for its time complexity. This section focuses on the derivation of the space and time expressions that are upper bounds for the space and time complexity of *Parallel FastLSA*.

3.3.1 *FastLSA* Recursion Pattern

In order to compute the amount of space and time required by *Parallel FastLSA* to align a sequence of size m against a sequence of size n using a FastLSA Grid Cache of size k, one needs to know the *trace* of the *FastLSA* algorithm. A trace of *FastLSA* is a series of FastLSA subproblems solved by the recursive calls to *FastLSA*, and which are listed in the exact order in which they are solved. A typical series for $PFastLSA(m,n,k)$ is:

$$\begin{aligned} PFastLSA(m,n,k) = PFillCache(m,n,k), PFastLSA(\tfrac{m}{k},\tfrac{n}{k},k),\\ PFastLSA(m_1,n_1,k),\ldots,PFastLSA(m_z,n_z,k); \end{aligned} \tag{3.1}$$

where $PFillCache(m,n,k)$ is the initial Fill Cache subproblem, $PFastLSA(\frac{m}{k},\frac{n}{k},k)$ is the recursive call to the bottom-right subproblem, and $PFastLSA(m_i,n_i,k)$, $i=1,z$ are the subproblems solved recursively inside the `while`-loop of the algorithm (i.e., the call in line 13 from Figure 3.1). Depending on the configuration of the optimal alignment path that is followed by the *FastLSA* algorithm, z can take values between $k-1$ and $2k-2$. Details about the values of z in the best case and worst case scenarios can be found in [Charter *et al.*, 2000]. Comprehensive analysis of the time and space complexity of the sequential *FastLSA* algorithm is presented in [Charter *et al.*, 2002].

Given a Base Case buffer of size BM, the deepest level of recursion reached by *FastLSA* is a positive integer, a, with

$$\frac{m}{k^a} \times \frac{n}{k^a} \leq BM < \frac{m}{k^{a-1}} \times \frac{n}{k^{a-1}}. \tag{3.2}$$

This is equivalent to

$$a - 1 < \frac{\log \frac{m \times n}{BM}}{2 \log k} \leq a \Leftrightarrow \left\lceil \frac{\log \frac{m \times n}{BM}}{2 \log k} \right\rceil = a. \tag{3.3}$$

3.3.2 Space Complexity

Definition 1 *Let $S(m, n, k)$ be the maximum number of d.p. matrix entries that need to be stored in order to align a sequence of size m against a sequence of size n using a grid cache with k rows and k columns.*

The following result shows that $S(m, n, k)$ is linear in m and n.

Theorem 2 *Let $S(m, n, k)$ be defined as in Definition 1. If the tiles for each Fill Cache subproblem are laid out in R rows and C columns, then*

$$S(m, n, k) \leq (3k - 1) \times (m + n) + \frac{P}{C} \times n + R \times C - u \times v + BM. \tag{3.4}$$

Proof. For an algorithm trace such as that in Equation 3.1,

$$\begin{aligned} S(m, n, k) &= \text{maxSpace}(PFastLSA(m, n, k)) \\ &= \max\Big(\text{maxSpace}(PFillCache(m, n, k)), \\ &\quad GridSpace(m, n, k) + \text{maxSpace}(PFastLSA(\tfrac{m}{k}, \tfrac{n}{k}, k)), \\ &\quad GridSpace(m, n, k) + \text{maxSpace}(PFastLSA(m_1, n_1, k)), \ldots, \\ &\quad GridSpace(m, n, k) + \text{maxSpace}(PFastLSA(m_z, n_z, k))\Big). \end{aligned} \tag{3.5}$$

Because $\frac{m}{k} \geq m_i$ and $\frac{n}{k} \geq n_i$, $\forall i, 1 \leq i \leq z$, the following is true:

$$\text{maxSpace}(PFastLSA(\tfrac{m}{k}, \tfrac{n}{k}, k)) \geq \text{maxSpace}(PFastLSA(m_i, n_i, k)), \forall i, 1 \leq i \leq z. \tag{3.6}$$

Equation 3.5 becomes

$$\begin{aligned} S(m, n, k) = \max\Big(&\text{maxSpace}(PFillCache(m, n, k)), \\ &GridSpace(m, n, k) + \text{maxSpace}(PFastLSA(\tfrac{m}{k}, \tfrac{n}{k}, k))\Big) \\ = \max\Big(&\text{maxSpace}(PFillCache(m, n, k)), GridSpace(m, n, k) + S(\tfrac{m}{k}, \tfrac{n}{k}, k)\Big). \end{aligned} \tag{3.7}$$

For the current implementation of the *Parallel FastLSA* algorithm, $PFillCache(m,n,k)$ uses $(k-1)(m+n)$ entries to store the local copy of the FastLSA Grid Cache, $(k-1)(m+n)$ entries to store the global, shared copy of the Grid Cache, $m+n$ entries to store the Tile Cache, $R \times C - u \times v$ entries to store the upper-left corner of each tile, and $\frac{n}{C}$ entries on each processor for computing a tile using a modified version of the *LastRow* algorithm. In summary,

$$\begin{aligned}\text{maxSpace}(PFillCache(m,n,k)) = A(m,n,k) &= (k-1)(m+n) + (k-1)(m+n) + \\ &+ (m+n) + R \times C - u \times v + P\tfrac{n}{C} \\ &= (2k-1)(m+n) + R \times C - u \times v + \tfrac{P}{C}n,\end{aligned} \tag{3.8}$$

and

$$GridSpace(m,n,k) = (k-1)(m+n). \tag{3.9}$$

Using the previous two results, Equation 3.7 becomes

$$\begin{aligned}S(m,n,k) &= \max\Big(A(m,n,k), (k-1)(m+n) + S(\tfrac{m}{k}, \tfrac{n}{k}, k)\Big) \\ &= \max\Big(A(m,n,k), (k-1)(m+n) + \\ &+ \max\Big(A(\tfrac{m}{k}, \tfrac{n}{k}, k), (k-1)\tfrac{m+n}{k} + S(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k)\Big)\Big) \\ &= \max\Big(A(m,n,k), \max\Big((k-1)(m+n) + A(\tfrac{m}{k}, \tfrac{n}{k}, k), \\ &(k-1)(m+n) + (k-1)\tfrac{m+n}{k} + S(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k)\Big)\Big) \\ &= \max\Big(\max\Big(A(m,n,k), (k-1)(m+n) + A(\tfrac{m}{k}, \tfrac{n}{k}, k)\Big), \\ &(k-1)(m+n)(1+\tfrac{1}{k}) + S(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k)\Big).\end{aligned} \tag{3.10}$$

In order to unwind the recursive formula from Equation 3.10, the result of Lemma 3 is used. Note that the result and the proof of Lemma 3 are given immediately after the proof of Theorem 2. Lemma 3 states that if $A(m,n,k)$ is defined as in Equation 3.8, then

$$A(m,n,k) \geq (k-1)(m+n)(1 + \tfrac{1}{k} + \cdots + \tfrac{1}{k^{j-2}}) + A(\tfrac{m}{k^{j-1}}, \tfrac{n}{k^{j-1}}, k), \forall j, 2 \leq j \leq a. \tag{3.11}$$

For example, for $j=2$, the inequality of Lemma 3,

$$A(m,n,k) \geq (k-1)(m+n) + A(\tfrac{m}{k}, \tfrac{n}{k}, k), \tag{3.12}$$

can be rewritten as

$$\max\Big(A(m,n,k), (k-1)(m+n) + A(\tfrac{m}{k}, \tfrac{n}{k}, k)\Big) = A(m,n,k). \tag{3.13}$$

By rewriting the inequalities of Lemma 3 for every value of j, exactly as done for $j = 2$, and by using the resulting equalities at every step of the unwinding of the recursive relation, we obtain:

$$\begin{aligned} S(m,n,k) &= \max\Big(A(m,n,k), (k-1)(m+n)(1+\tfrac{1}{k}) + S(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k)\Big) \\ &= \cdots = \\ &= \max\Big(A(m,n,k), (k-1)(m+n)(1+\tfrac{1}{k}+\cdots+\tfrac{1}{k^{a-1}}) + S(\tfrac{m}{k^a}, \tfrac{n}{k^a}, k)\Big) \\ &= \max\Big(A(m,n,k), k(m+n)(1-\tfrac{1}{k^a}) + S(\tfrac{m}{k^a}, \tfrac{n}{k^a}, k)\Big). \end{aligned} \tag{3.14}$$

Because $PFastLSA(\frac{m}{k^a}, \frac{n}{k^a}, k)$ is a Base Case subproblem, $S(\frac{m}{k^a}, \frac{n}{k^a}, k) \leq BM$; thus, $S(m,n,k)$ is bounded above by

$$\begin{aligned} \max\Big(A(m,n,k), k(m+n)(1-\tfrac{1}{k^a}) + BM\Big) &\leq A(m,n,k) + k(m+n)(1-\tfrac{1}{k^a}) + BM \\ &\leq A(m,n,k) + k(m+n) + BM = \\ &= (2k-1)(m+n) + R \times C - u \times v + \\ &+ \tfrac{P}{C}n + k(m+n) + BM \\ &= (3k-1) \times (m+n) + \tfrac{P}{C}n + \\ &+ R \times C - u \times v + BM. \end{aligned} \tag{3.15}$$

Therefore,

$$S(m,n,k) \leq (3k-1) \times (m+n) + \tfrac{P}{C} \times n + R \times C - u \times v + BM, \tag{3.16}$$

which concludes the proof of Theorem 2. ■

Lemma 3 *Let $A(m,n,k)$ be defined as in Equation 3.8. Then*

$$A(m,n,k) \geq (k-1)(m+n)(1+\tfrac{1}{k}+\cdots+\tfrac{1}{k^{j-2}}) + A(\tfrac{m}{k^{j-1}}, \tfrac{n}{k^{j-1}}, k), \forall j, 2 \leq j \leq a. \tag{3.17}$$

Proof. Let j be such that $2 \leq j \leq a$. The inequality becomes

$$\begin{aligned} A(m,n,k) \geq (k-1)(m+n)(1+\tfrac{1}{k}+\cdots+\tfrac{1}{k^{j-2}}) + A(\tfrac{m}{k^{j-1}}, \tfrac{n}{k^{j-1}}, k) &\Leftrightarrow \\ A(m,n,k) - A(\tfrac{m}{k^{j-1}}, \tfrac{n}{k^{j-1}}, k) \geq k(m+n)(1-\tfrac{1}{k^{j-1}}) &\Leftrightarrow \\ (2k-1)(m+n)(1-\tfrac{1}{k^{j-1}}) + \tfrac{P}{C}n(1-\tfrac{1}{k^{j-1}}) \geq k(m+n)(1-\tfrac{1}{k^{j-1}})&. \end{aligned} \tag{3.18}$$

Because $\frac{P}{C}n(1-\frac{1}{k^{j-1}}) \geq 0$, it is sufficient to prove that

$$\begin{aligned} (2k-1)(m+n)(1-\tfrac{1}{k^{j-1}}) \geq k(m+n)(1-\tfrac{1}{k^{j-1}}) &\Leftrightarrow \\ (2k-1) \geq k &\Leftrightarrow \\ k \geq 1&, \end{aligned} \tag{3.19}$$

which is true. Therefore, the inequality of Lemma 3 is true $\forall j, 2 \leq j \leq a$. ■

3.3.3 Time Complexity

Definition 4 *Let $WT(m,n,k,P)$ be the time spent by the slowest of the P threads involved in the parallel alignment of two sequences of size m and n, using a grid cache with k rows and k columns.*

The time spent by the slowest thread, $WT(m,n,k,P)$, is a good upper bound for the time complexity of *Parallel FastLSA*. An upper bound for $WT(m,n,k,P)$ itself is established by the following result.

Theorem 5 *Let $WT(m,n,k,P)$ be defined as in Definition 4. For simplicity, assume that the tiles processed in a parallel phase are laid out in R rows and C columns for both the Fill Cache and the Base Case subproblems. Then*

$$WT(m,n,k,P) \leq \tfrac{m \times n}{P} \times (1 + \tfrac{P^2-P}{R \times C}) \times (\tfrac{k}{k-1})^2. \tag{3.20}$$

Proof. Let $PFillCacheT(M,N,k,P)$ be the time spent by the slowest of the P threads when solving a Fill Cache subproblem of size $M \times N$. From the definition of $WT(m,n,k,P)$ and that of a trace of the *FastLSA* algorithm (i.e., Equation 3.1), it can be inferred that

$$WT(m,n,k,P) = PFillCacheT(m,n,k,P) + (2k-1) \times WT(\tfrac{m}{k}, \tfrac{n}{k}, k, P). \tag{3.21}$$

The first step of the proof is to find a good approximation for $PFillCacheT(M,N,k,P)$. As explained in Section 3.1, the d.p. matrix entries that are computed in order to fill the Grid Cache are partitioned in $R \times C - u \times v$ tiles. Some of the tiles can be empty, so this number is actually an upper bound. If the Fill Cache subproblem has M rows and N columns, each tile has at most $\frac{M}{R} \times \frac{N}{C}$ entries. Let T be the time spent by one processor to compute a tile sequentially. Because each tile is solved using the *LastRow* algorithm from Figure 2.3, we have $T = O(\frac{M \times N}{R \times C})$.

As shown in Figure 3.4, the computation of the tiles advances following a diagonal wavefront pattern. In Figure 3.4, each diagonal of tiles labeled with the same number forms a *wavefront line.* A wavefront line is important because the tiles that form it are independent and can be computed in parallel.

The computation of the tiles for a Fill Cache subproblem can be divided into three distinct phases. Figure 3.6 shows the three phases corresponding to a Fill Cache subproblem which is solved on $P = 8$ processors, using $k = 6$, $u = 2$, and $v = 3$. Each wavefront line is labeled with the number of tiles that form that particular wavefront line. A good approximation for $PFillCacheT(M,N,k,P)$ can be found using an upper bound for the time spent in each phase.

In the first phase, the number of tiles in each wavefront line increases from 1 to $P-1$. In this phase, a total of $\frac{P(P-1)}{2}$ tiles are computed. In the worst case scenario, each wavefront line is solved in a parallel stage that lasts a time of T; thus, the time spent on the first phase is at most $(P-1)T$.

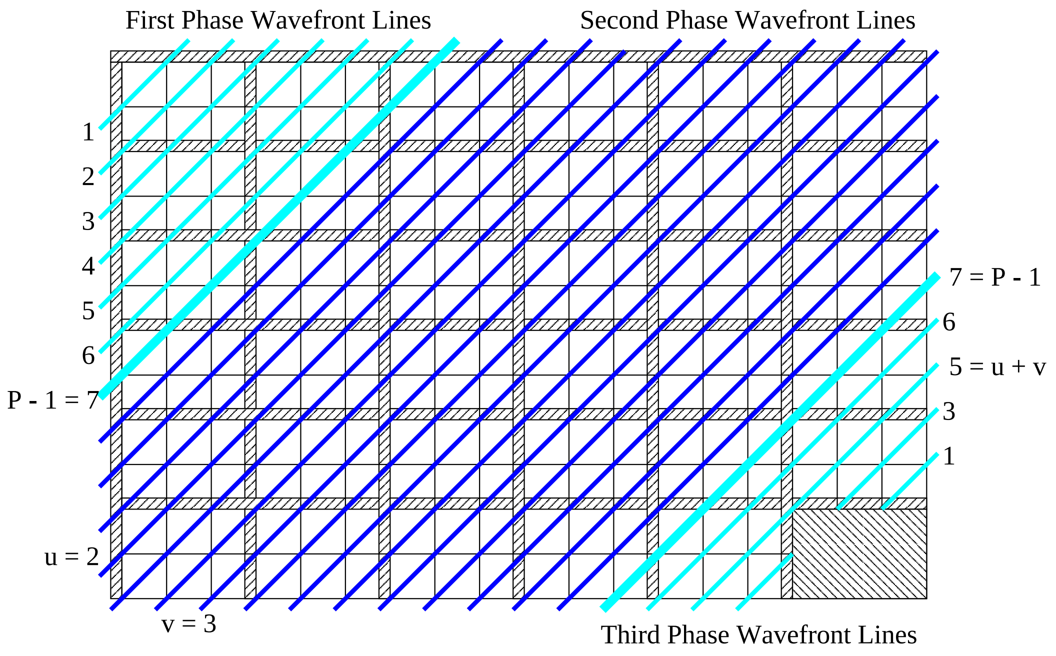

First Phase Wavefront Lines
Second Phase Wavefront Lines
1
2
3
4
5
6
P - 1 = 7
u = 2
v = 3
7 = P - 1
6
5 = u + v
3
1
Third Phase Wavefront Lines

Figure 3.6: The Three Phases of a Parallel Fill Cache Subproblem

The third phase consists of the wavefront lines that are formed from less than P tiles and that are not computed in the first phase. An example of wavefront lines forming a third phase is depicted in Figure 3.6. Some of the wavefront lines of this phase may not consist of contiguous tiles because the tiles belonging to the bottom-right FastLSA subproblem are not computed for a Fill Cache subproblem (e.g., the wavefront line labeled 3 in Figure 3.6).

The third phase has at most the same number of wavefront lines as the first phase, i.e., $P-1$. Because each wavefront line can be solved in a parallel stage of time T, the third phase cannot last longer than $(P-1)T$. The number of tiles that are computed in the third phase is difficult to estimate for general values of P, u, and v, but a lower bound for this number is $\frac{P(P-1)}{2} - u \times v$.

The second phase is the true parallel phase. Enough tiles are available so that all processors can work in parallel. An upper bound for the number of tiles computed in this phase is the total number of tiles, minus the number of tiles computed in the first phase and the lower bound for the number of tiles computed in the third phase, i.e.,

$$(R \times C - u \times v) - \frac{P(P-1)}{2} - \left(\frac{P(P-1)}{2} - u \times v\right) = R \times C - P^2 + P. \qquad (3.22)$$

Because these tiles are computed in parallel, the time spent in the second phase is

$$\frac{(R \times C - P^2 + P)}{P} \times T. \qquad (3.23)$$

Note that we need a lower bound for the number of tiles computed in the third phase in order to compute an upper bound for the time spent in the second phase.

An approximation for $PFillCacheT(M, N, k, P)$ is obtained through the summation of the times for the three phases, which gives

$$\begin{aligned} PFillCacheT(M,N,k,P) &= (P-1)T + \tfrac{(R\times C-P^2+P)}{P}T + (P-1)T \\ &= \tfrac{(R\times C+P^2-P)}{P}T \\ &= \tfrac{(R\times C+P^2-P)\times M\times N}{P\times R\times C} \\ &= M\times N\times \tfrac{1}{P}(1+\tfrac{P^2-P}{R\times C}) \\ &= M\times N\times\alpha, \end{aligned} \tag{3.24}$$

where

$$\alpha = \tfrac{1}{P}(1+\tfrac{P^2-P}{R\times C}). \tag{3.25}$$

Let $PBaseCaseT(M, N, P)$ be the time spent by the slowest of the P threads when solving a Base Case subproblem of size $M \times N$. An approximation for $PBaseCaseT(M, N, P)$ is obtained through a reasoning process similar to that used for $PFillCacheT(M, N, k, P)$. We get

$$\begin{aligned} PBaseCaseT(M,N,P) &= (P-1)T + (R\times C - \tfrac{P(P-1)}{2} - \tfrac{P(P-1)}{2})\tfrac{T}{P} + (P-1)T \\ &= (P-1)T + \tfrac{(R\times C-P^2+P)}{P}T + (P-1)T \\ &= M\times N\times\tfrac{1}{P}(1+\tfrac{P^2-P}{R\times C}) \\ &= M\times N\times\alpha. \end{aligned} \tag{3.26}$$

Using the results of Equation 3.24 and Equation 3.26, Formula 3.21 becomes

$$\begin{aligned} WT(m,n,k,P) &= m\times n\times\alpha + (2k-1)\times WT(\tfrac{m}{k},\tfrac{n}{k},k,P) \\ &= mn\alpha + (2k-1)(\tfrac{m}{k}\tfrac{n}{k}\alpha + (2k-1)WT(\tfrac{m}{k^2},\tfrac{n}{k^2},k,P)) \\ &= mn\alpha + mn\alpha\tfrac{2k-1}{k^2} + (2k-1)^2 WT(\tfrac{m}{k^2},\tfrac{n}{k^2},k,P) \\ &= mn\alpha + mn\alpha\tfrac{2k-1}{k^2} + mn\alpha(\tfrac{2k-1}{k^2})^2 + (2k-1)^3 WT(\tfrac{m}{k^3},\tfrac{n}{k^3},k,P) \\ &= \cdots = \\ &= mn\alpha(1+\tfrac{2k-1}{k^2}+(\tfrac{2k-1}{k^2})^2+\cdots+(\tfrac{2k-1}{k^2})^{a-1}) + (2k-1)^a WT(\tfrac{m}{k^a},\tfrac{n}{k^a},k,P) \\ &= mn\alpha(1+\tfrac{2k-1}{k^2}+\cdots+(\tfrac{2k-1}{k^2})^{a-1}) + (2k-1)^a PBaseCaseT(\tfrac{m}{k^a},\tfrac{n}{k^a},P) \\ &= mn\alpha(1+\tfrac{2k-1}{k^2}+\cdots+(\tfrac{2k-1}{k^2})^{a-1}) + (2k-1)^a \tfrac{m}{k^a}\tfrac{n}{k^a}\alpha \\ &= mn\alpha(1+\tfrac{2k-1}{k^2}+\cdots+(\tfrac{2k-1}{k^2})^{a-1}+(\tfrac{2k-1}{k^2})^{a}) \\ &= mn\alpha\frac{1-(\frac{2k-1}{k^2})^{a+1}}{1-\frac{2k-1}{k^2}}. \end{aligned} \tag{3.27}$$

Because $(\frac{2k-1}{k^2})^{a+1} > 0$, we have

$$\begin{aligned} WT(m,n,k,P) &= mn\alpha\frac{1-(\frac{2k-1}{k^2})^{a+1}}{1-\frac{2k-1}{k^2}} \\ &\le mn\alpha\frac{1}{1-\frac{2k-1}{k^2}} = mn\alpha(\tfrac{k}{k-1})^2. \end{aligned} \tag{3.28}$$

By replacing α with its value (Equation 3.25), it becomes true that

$$WT(m,n,k,P) \leq mn\alpha(\tfrac{k}{k-1})^2 = \tfrac{m\times n}{P} \times (1 + \tfrac{P^2-P}{R\times C}) \times (\tfrac{k}{k-1})^2, \tag{3.29}$$

which concludes the proof of Theorem 5. ■

3.4 Concluding Remarks

This chapter introduces *Parallel FastLSA*, which is a new parallel algorithm for optimal pairwise sequence alignment. We present a detailed description of *Parallel FastLSA* and explain the wavefront parallelism that is used by the algorithm. The description of *Parallel FastLSA* includes details of the logical structures involved in the parallel computation of each of the two types of FastLSA subproblems, Fill Cache and Base Case.

We also provide details of the two strategies that we use to allocate parallel tasks to processors. In the first strategy, Dynamic Distribution of Work, the tiles that are ready for processing are placed in a queue from where the processors dynamically dequeue them. In the second strategy, Static Distribution of Work, entire rows of tiles are preassigned to processors in a round-robin fashion. Each of these strategies leads to a different implementation for *Parallel FastLSA*.

We show that *Parallel FastLSA* still has linear space complexity by deriving a linear upper bound for the maximum number of d.p. matrix entries that need to be stored at any time during the execution of the algorithm (Theorem 2). We also derive an upper bound for the maximum number of d.p. matrix entries that are computed by *Parallel FastLSA*, and conclude that its time complexity is still quadratic (Theorem 5).

Chapter 4

Experimental results for *Parallel FastLSA*

The previous chapter provides upper bounds for the space and time complexity of *Parallel FastLSA*. Although these results show what type of curve the space and time requirements of *Parallel FastLSA* follow, they do not show that good speedups can be achieved in practice when running *Parallel FastLSA* on P processors.

Because of this drawback of the theoretical analysis, we have run a large number of experiments in order to assess the empirical efficiency of *Parallel FastLSA*. Our experiments with *Parallel FastLSA* show good speedups, especially when long sequences are aligned. The speedups are almost linear for 8 processors or less. This chapter describes some of the experiments performed, and explains in detail the results obtained.

4.1 Experimental Methodology

We present results from the experiments we have performed with *Parallel FastLSA* on an SGI Origin 2400 parallel computer. The Origin 2400 has 64 processors (400 MHz R12000 MIPS CPUs), each with a primary data cache of 32 KBytes and a unified 8 MB secondary cache. The *Parallel FastLSA* algorithm is implemented in `C` using Irix 6.5 `sproc` threads with hardware-based shared memory. The sequential version of the *FastLSA* algorithm is an independent, non-commercial implementation based on the original description [Charter *et al.*, 2000] and discussions with the designers of the algorithm. The *FastLSA* implementations that we benchmark find the globally optimal alignment of two sequences using the straightforward scoring function discussed in Section 1.3 (i.e., Match = 2, Mismatch = -1, Gap_Penalty = -2).

This chapter discusses in detail the experimental results corresponding to the alignment of three pairs of DNA sequences which are chosen from a test suite suggested by the bioinformatics group at Penn State University [Penn State University, 2001*a*]. Most of their examples are comparisons of "some region of the human genome with the synthenic region from a rodent genome" [Penn State University, 2001*c*]. We feel that it is important to apply *Parallel FastLSA* to real life examples. These pairs are considered as a test suite, not only because of their size, but also because their alignment is biologically meaningful. Although we have experimented with several more pairs of DNA sequences, we choose to present results for the pairs of shortest and longest sequences, and another pair of sequences of medium size.

1. The shortest sequence pair is formed by the *XRCC1* DNA repair gene from human beings and mice. The *XRCC1* gene encodes an enzyme involved in the repair of X-ray damage [Penn State University, 2001*c*]. The human sequence is 37,785 bp long, and the mouse sequence is 37,349 bp long.

2. The medium size sequences are the "cardiac myosin heavy chain genes" (abbreviated *Myosin*) [Penn State University, 2001*c*] from human beings and hamsters. The human sequence is 55,820 bp long, and the hamster sequence is 66,315 bp long.

3. The longest sequence pair consists of the human and mouse alpha/delta T-cell receptor loci (abbreviated *TCR*). These sequences "show an unusually high level of conservation" [Penn State University, 2001*b*]. The human sequence is 319,030 bp long, and the mouse sequence is 305,636 bp long.

Throughout the benchmarking process discussed in this chapter, all parameters introduced in Section 3.1 are assigned constant, empirical values. We opt for this solution because *Parallel FastLSA* involves eight parameters that can vary, and tuning all of them is a complicated task. Choosing empirical values for the parameters is justified by the fact that we are interested in establishing reasonable performance for *Parallel FastLSA* rather than optimal performance. In the future, we hope to further explore the parameter space.

Table 4.1 summarizes the parameters involved in the *FastLSA* algorithms and the values assigned to them. After running a series of experiments with different values for u, v and k we restricted ourselves to these empirically validated values. These values are deemed to provide the *FastLSA* algorithms with the opportunity to run reasonably fast. In particular, *Parallel FastLSA* is run with $R = 8$, $C = 10$ for the Base Case subproblems, and $u = 3$, $v = 4$ (i.e., $R = 3 \times k$, $C = 4 \times k$) for the Fill Cache subproblems. These preset values are used for each FastLSA subproblem, independently of its size or level of the recursion.

The only parameters which vary during the benchmarking process are k and the size of the sequences aligned. The parameter k iterates from 8 to 12 in order to assess the impact which the size of the FastLSA Grid Cache has on the performance of the algorithm. The Base Case buffer size, BM, is assigned the constant value of $1,600,000$. Note that these last parameters influence the performance of both the sequential and the parallel versions of FastLSA.

The parameter values that we have chosen for u, v, and k are non-optimal for $P = 32$, and the explanation of this fact follows. The logical d.p. matrix is divided in $3 \times k$ rows and

	Parameter Name	**Parameter Value**	**Notes**
Constant	u	3	number of rows of tiles between consecutive Grid rows;
	v	4	number of columns of tiles between consecutive Grid columns;
	BM	1,600,000	size of Base Case buffer in integers;
	R	8	total number of rows of tiles for a Base Case subproblem;
	C	10	total number of rows of tiles for a Base Case subproblem;
Variable	P	1, 2, 4, 8, 16, 32	number of processors;
	k	8–12	number of Grid rows and columns;
	R	$3 \times k$	total number of rows of tiles for a Fill Cache subproblem;
	C	$4 \times k$	total number of rows of tiles for a Fill Cache subproblem;
	size of d.p. matrix	$37,349 \times 37,785$	*XRCC1*;
		$55,820 \times 66,315$	*Myosin*;
		$305,636 \times 319,030$	*TCR*.

Table 4.1: The Parameters which Influence the *FastLSA* algorithms

$4 \times k$ columns of tiles for each Fill Cache subproblem. Because the wavefront line can have no more tiles than the shortest dimension of the array of tiles, the wavefront line can have at most $3 \times k$ tiles for our parameter values. When k is less than 11, the wavefront line consists or less than 32 tiles, which means that 32 processors cannot all work in parallel. Despite this theoretical disadvantage, we observed that, for $P = 32$, $k = 8$ is the empirical optimum for the alignment of the *XRCC1* sequences, while $k = 9$ is the empirical optimum for the *Myosin* sequences.

The performance results for *Parallel FastLSA* presented in this chapter are obtained using an implementation based on the Dynamic Distribution of Work strategy. This strategy of work distribution is introduced in Subsection 3.2.1. We have also benchmarked an implementation based on the Static Distribution of Work strategy, but choose not to present separate results for it because they are similar to those obtained for the implementation based on the Dynamic Distribution of Work strategy.

The version of *Parallel FastLSA* analyzed in this chapter solves the Base Case subproblems sequentially. This modified version of *Parallel FastLSA* is preferred to the one described in Section 3.1 because of its better performance. The performance numbers show that solving the Base Case subproblems in parallel is consistently and considerably slower than solving them sequentially. The comparison is made between the total time spent on solving Base Case subproblems by *Parallel FastLSA* and the sequential *FastLSA*. Our intuition is that the Base Case subproblems are too small to benefit from parallelism. Section 4.7 gives a clear picture that the version of *Parallel FastLSA* that solves the Base Case subproblems sequentially outperforms the initial version, which solves the Base Case subproblems in parallel.

The SGI machine used to benchmark *FastLSA*, both sequential and parallel, can be accessed only through a batch queueing and workload management system (Portable Batch System [Veridian Systems, 2001]). Although the SGI Origin is a multiprogrammed computer, the performance numbers are quite stable from one execution to the other. In order to remove the small, unpredictable noise generated by the operating system, three consecutive runs are performed for each set of parameter values which is benchmarked. The three time samples obtained for each run are averaged.

The performance of the *FastLSA* algorithms is optionally instrumented by recording relevant trace information during their execution. The total execution time, the total time spent on each FastLSA subproblem, the type of each subproblem and its coordinates in the initial d.p. matrix are saved in a trace file created for every combination of P and sequence pairs. In addition to the above information, for every Fill Cache subproblem, *Parallel FastLSA* also records per thread information such as the time for computing a tile and the time spent at the barrier that follows the parallel region. All the graphs and tables presented in this chapter are generated by processing the information collected in the trace files. Because the trace collecting mechanism was always on, the total execution times shown here may be slightly bigger than in reality.

4.2 General Observations

As mentioned in the previous section, the sequential and parallel versions of *FastLSA* are benchmarked for each value of k from 8 to 12, and for each of the three pairs of sequences. Ideally, we should have devised a simple, reliable heuristic which produces an optimal value for k, given the size of the sequences and P, the number of processors used. This optimal value would ensure that the overall alignment time is in a close vicinity of the theoretical optimal time. However, the relationship between the optimal value of k, P, and the size of the sequences is not straightforward, and this makes the development of such a heuristic challenging. We note from the results obtained that, in most of the cases, there is a small number of neighboring values that can be chosen as empirically optimal values for k. The values outside this small interval, when assigned to k, worsen the time performance of the algorithm. The 8 to 12 interval for k was chosen after repeated probing for optimal values. This interval includes an empirical optimal value for k in most of the combinations benchmarked.

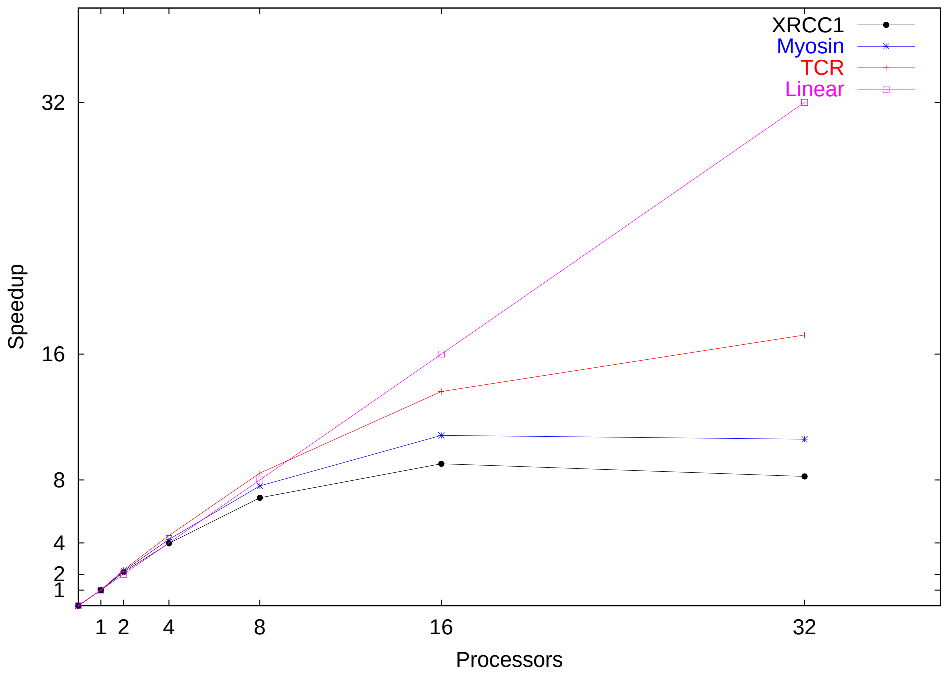


Figure 4.1: Best Speedups for XRCC1, Myosin, and TCR

In order to simulate the effect of such a heuristic on the time performance of *Parallel FastLSA* and to provide a quick first look into the results of our experiments, we have selected for each pair of sequences and each number of processors the best execution time across the five values of k that were considered, and then computed the speedups. The resulting speedup curves are shown in Figure 4.1. Table 4.2 shows the empirically optimal execution time for each sequence alignment performed and the corresponding empirically optimal value for k.

For the pair of short sequences, *XRCC1*, the speedup is linear for 2 and 4 processors, but starts deteriorating when 8 or more processors are used. The slowdown from 16 and 32 processors occurs because the granularity of the work assigned to each processor decreases, leading to a situation where the processors spend more time trying to get a tile on which to work rather than actually working on it.

The speedup curve for the alignment of the *Myosin* sequences ascends almost linearly for up to 8 processors, increases slowly for 16 processors, and almost flattens for 32 processors. This noticeable improvement of the performance of *Parallel FastLSA* happens because the d.p. matrix computed for the *Myosin* sequences has 2.6 times more entries than the d.p. matrix computed for the *XRCC1* sequences. The larger *Myosin* d.p. matrix provides better granularity for the parallel tasks, but not enough to satisfy 32 processors.

The best speedup curve is obtained for the largest sequences that are aligned. As mentioned above, both *TCR* sequences are over 300,000 base pairs in length. Because of the large problem, the granularity of work is reasonable and the speedup becomes slightly

Sequences	Number of Processors	Time (sec.)	Speedup	Optimal k
XRCC1	1	71.71		12
	2	33.44	2.14	11
	4	18.05	3.97	10
	8	10.44	6.87	9
	16	7.94	9.03	9
	32	8.72	8.22	8
Myosin	1	189.71		12
	2	85.54	2.22	12
	4	44.92	4.22	11
	8	24.89	7.62	11
	16	17.52	10.83	11
	32	17.91	10.59	9
TCR	1	5040.93		12
	2	2202.65	2.29	12
	4	1128.56	4.47	12
	8	597.66	8.43	12
	16	370.07	13.62	12
	32	292.84	17.21	12

Table 4.2: Empirically Optimal k, Execution Times, and Speedups

super-linear for 8 processors or less. The super-linearity of the speedup is due to cache effects, which are a reality of any ccNUMA architecture, including the SGI Origin [Laudon and Lenoski, 1997].

For 2 to 8 processors, the time spent by *Parallel FastLSA* on servicing memory accesses is significantly lower than that spent by the sequential *FastLSA*. The decrease of memory access time, coupled with the decrease in computing time caused by parallelism, makes the speedup of the *Parallel FastLSA* super-linear. Specifically, the memory requirements for aligning the *TCR* sequences are substantial for at least the initial Fill Cache subproblem. When the sequences are aligned on only one processor, the L2 data cache is not large enough to hold all the data accessed by the processor, and large numbers of data cache misses are incurred. Because the FastLSA Grid Cache is allocated at different points in the ccNUMA architecture, servicing data cache misses can be expensive. When the sequences are aligned in parallel, there are better chances that the cache of each processor can hold its working data set.

The speedup curve for *TCR* is steeper from 8 to 16 processors than the speedup for *Myosin*, and a reasonable improvement of the performance occurs for 32 processors. The speedup curve increases from 16 to 32 processors with a slope of 0.22 – which is close to 0.27, the slope of the speedup curve for *XRCC1* between 8 and 16 processors.

In our experiments, we have also found that the majority of the alignment time is spent solving the initial Fill Cache subproblem. For each alignment operation performed by *Parallel FastLSA*, we computed the percentage of time spent on the initial Fill Cache subproblem, out of the total execution time. For the *TCR* pair, this percentage ranges from 87.86% for $P = 1$ to 77.08% for $P = 16$, and 67.53% for $P = 32$. We note that the above defined percentage decreases with P, but increases with the size of the sequences; for $P = 16$, the percentage is 59.03% for *XRCC1* and 63.40% for *Myosin*. Because of the design of the FastLSA algorithms, the time spent on the initial Fill Cache subproblem depends only on the size of the sequences, and not their particular configuration.

4.3 Description of Graph Types

In order to understand how the parameters and the design of *Parallel FastLSA* influence its execution time, the remainder of this chapter is dedicated to a detailed empirical analysis of the performance of the algorithm. For each pair of sequences aligned, the same series of graphs is presented in order to facilitate cross-comparison. Most of these graphs present performance numbers which are broken down in an attempt to approximate the distribution of the time spent at different stages of an alignment.

Recall from Section 2.3 that *FastLSA* solves a succession of rectangular problems, called FastLSA subproblems, using either a Base Case approach for the small subproblems, or a General Case approach for the subproblems that do not fit in the Base Case buffer. *Parallel FastLSA* solves the Base Case subproblems sequentially in the current implementation, but it fills the FastLSA Grid Cache in parallel for the Fill Cache subproblems (Section 3.1).

The time spent by the *FastLSA* algorithms computing a pairwise alignment is primarily determined by the total time spent by the algorithms on filling matrices for Base Case subproblems, or filling Grid Caches for Fill Cache subproblems. Ideally, we would like to present the time spent on each subproblem, and the breakdown times for each subproblem. Unfortunately, because there are thousands of these subproblems for each sequence pair, an approximation of the statistical distribution of the subproblem execution times is preferred. The subproblems are clustered together based on the type or size of the subproblem, and the execution times are accumulated for the subproblems inside each resulting partition set. The clustering is done by processing the trace files, and the graphs obtained are presented in the following three sections.

4.3.1 Subproblem Count Graph

Each series of graphs begins with a subproblem count graph which shows how many FastLSA subproblems are solved during an alignment operation, and how large these problems are. Note that the FastLSA subproblems which occur for a FastLSA alignment are determined by the sequences, the size of the Base Case buffer and k, and are independent of the number of processors used for the alignment. Therefore, only one counting graph is shown for each pair of aligned sequences. This graph (e.g., Figure 4.2) consists of two plots: one for the clustering based on the type of the subproblems, and the other for the clustering based on the size of the subproblems.

Note that the clustering process described here is applied identically for all the metrics benchmarked and reported in this chapter. For the clustering by type, the subproblems are assigned to the Base Case set or the Fill Cache set according to their type. The plot for the type-based clustering presents the number of subproblems in each of the two partition sets for the five different values of k. The black bar on the left side of each pair of solid bars represents the number of Base Case subproblems, whereas the red bar on the right represents the number of Fill Cache subproblems.

The clustering by size is a further refinement of the type-based clustering. The Base Case subproblems are distributed into three partition subsets based on their size (i.e., number of d.p. matrix entries). The first partition holds the smallest subproblems, up to $\frac{1}{3}BM$ in size; the second partition holds those between $\frac{1}{3}BM$ and $\frac{2}{3}BM$; the third holds the biggest ones, sized up to and including BM. For Fill Cache subproblems, the interval between BM and the size of the initial d.p. matrix is evenly divided into five subintervals. Each subinterval is assigned a partition subset to which a Fill Cache subproblem is distributed if its size falls within that subinterval. The result is a cluster with three partition subsets for Base Case and five partition subsets for Fill Cache.

The plot for the size-based partitioning shows eight bars for each value of k. The three black bars on the left indicate the number of FastLSA subproblems in the Base Case partitions, while the five red bars to the right indicate the number of FastLSA subproblems in the Fill Cache partitions. The five groups of bars are separated by thin, vertical, blue lines which are used only as dividers.

4.3.2 Execution Time Graphs

Execution time is one of the most important indicators of the performance of an algorithm. Two graphs presenting the execution times for the FastLSA alignments are next in the series of graphs that we lay out for each sequence pair. One of the two graphs shows the execution times clustered based on the type of the subproblems, while the other shows the execution times clustered based on the size of the subproblems. Each of the two graphs shows six plots corresponding to the number of processors used for benchmarking. In the trace file, the execution time is measured overall for the alignment operation and individually for each FastLSA subproblem.

When the FastLSA subproblems are clustered based on their type, the time is added separately for the Base Case subproblems and the Fill Cache subproblems. The results are shown in each plot as stacked bars, with each stack corresponding to a value of k (e.g., Figure 4.3). The cumulative time spent solving Base Case subproblems is shown as a black bar, and above it, there is a red bar representing the cumulative time spent on Fill Cache subproblems. The remaining time to the total time of the alignment is depicted as a blue-filled bar which is stacked at the top.

For the size-based clustering, the times recorded for each subproblem are added separately for each of the eight partition sets. The results are shown in each plot as groups of filled bars, each group corresponding to a value of k (e.g., Figure 4.4). The three black bars on the left side of each group correspond to the cumulative time recorded for each of the three subsets of Base Case subproblems, while the five red bars on the right side represent the cumulative time for the five partition subsets of Fill Cache subproblems. The groups are separated by thin, vertical blue lines.

4.3.3 Speedup Graphs

Speedup is an accurate indicator of the effectiveness of *Parallel FastLSA*. An overall speedup is computed for every *Parallel FastLSA* alignment benchmarked, and breakdown speedups are computed for each partition set. The breakdown speedup for a partition set is computed by dividing the cumulative sequential time for that set by the cumulative parallel time for that same set.

Two graphs that show speedups for *Parallel FastLSA* follow the execution time graphs in the series. The first graph shows speedups for the clustering based on type; the second shows speedups for the clustering based on the size of the subproblems. Each of the two speedup graphs shows five plots which correspond to the five different numbers of processors used for benchmarking *Parallel FastLSA*.

The graph plots for the type-based clustering show the speedup for the set of Base Case subproblems as a black-filled bar, the speedup for the set of Fill Cache subproblems as a red-filled bar, and the overall speedup as a blue-filled bar. For each value of k, a group of three distinct filled bars is shown (e.g., Figure 4.5).

The graph plots for the size-based clustering have a layout similar to that of the graph plots for the size-based clustering of the execution time (e.g., Figure 4.6). For the partition sets that do not contain any FastLSA subproblems, the speedup is shown as 0.

4.3.4 Barrier Time Graphs

The series of graphs ends with two graphs that illustrate the delays that occur at the barriers following the parallel regions. These graphs give a measure of the delay incurred by each parallel region to the alignment process. A shorter delay at barriers is preferable.

The parallel region of a Fill Cache subproblem is the parallel filling of the Grid Cache. The barrier time is computed for each Fill Cache subproblem as the time elapsed between the arrival of the first thread at barrier, and the arrival of the last thread at barrier. Because each Base Case subproblem is solved sequentially by the master thread, we assume that this thread spends no time at the barrier that follows the subproblem, while the other threads wait as long as it takes the master thread to solve the subproblem.

The barrier delay for a Fill Cache subproblem is primarily due to the final stage of the parallel processing. This final stage is defined in the proof of Theorem 5 as the "third phase" of a parallel processing region. When the queue has no available tasks, and no more tasks

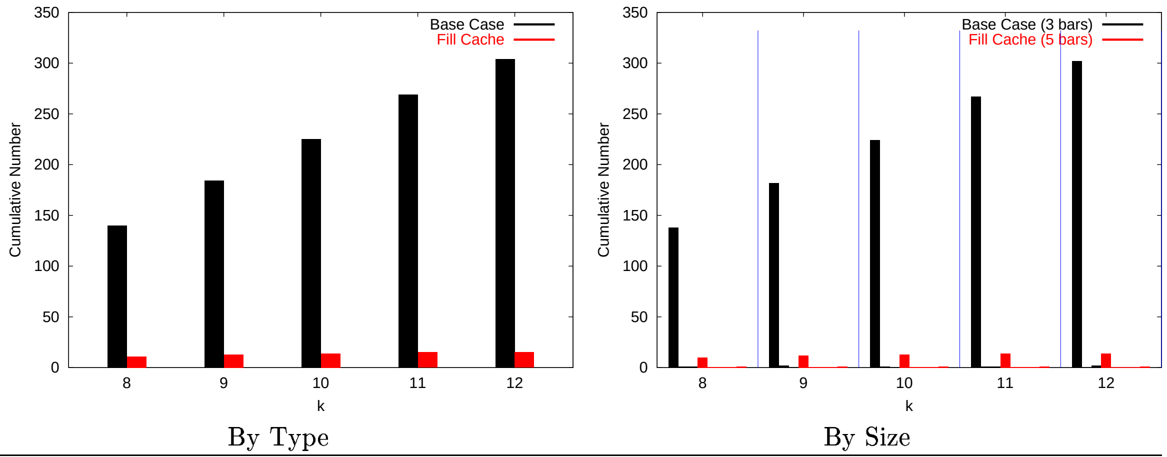


Figure 4.2: FastLSA Subproblem Count: *Parallel FastLSA* Alignment for Human *XRCC1* versus Mouse *XRCC1* (Breakdown Based on the Type/Size of the FastLSA Subproblems)

are expected to be enqueued, the first processor that asks for work is going to be the first that reaches the barrier. The other processors are still working on their tasks, or may have reserved in advance some of the tiles, and are now waiting for them to become available.

Theoretically, the barrier time cannot be higher than $P-1$ times the amount of time it takes a thread to process an entire tile. The worst case scenario happens when one threads reaches the barrier and the other $P-1$ threads work on their last tile, one after the other. This can happen if there is a chain dependency between the last $P-1$ tiles. The upper bound for the barrier time can be lowered by decreasing the granularity of the parallel tasks, but at the risk of reducing the speedup.

The graphs for barrier time have a layout similar to that of the graphs for execution time because they involve the overall time as well as a time value for each FastLSA subproblem. For example, in Figure 4.7 each plot of the type-based clustering graph shows the stacked values of the accumulated barrier time for Base Case and Fill Cache subproblems, and the remaining time to the overall time of the parallel alignment.

4.4 *XRCC1*

It can be seen from the subproblem count graph (Figure 4.2), that the number of Base Case subproblems is significantly larger than the number of Fill Cache subproblems, and this disparity increases with k. The Base Case buffer is quite large ($BM = 1,600,000$) and, at the third level of recursion, all the subproblems will be of Base Case type. Furthermore, each Fill Cache subproblem is partitioned into k^2 rectangles, and the FastLSA subproblems of the next recursion level are only the relevant portions of these rectangles. Even if a rectangle does not fit in the Base Case buffer, if the optimal path traverses only a small region of this rectangle that fits in the Base Case buffer, the resulting subproblem (the relevant region of the rectangle) is of Base Case type. The bigger the value of k, the smaller the rectangles in which the Fill Cache subproblems are partitioned, and the more Base Case subproblems will be generated.

In the plot that depicts the size-based clustering, the rightmost bar in each group is of size 1, but it is difficult to notice because of the scale of the plot. This means that the partition set that contains the biggest Fill Cache subproblems includes only one subproblem: the initial Fill Cache subproblem.

Figure 4.3 shows that the total execution time is optimal for $k = 12$ for the sequential alignment, and the optimal value for k decreases with the increase of P, the number of processors. For $P = 32$, the optimal value of k is 8. Figure 4.4 shows that the time spent on solving the initial Fill Cache subproblem decreases with the increase of P. However, for 16 and 32 processors, the time spent on the small-sized Fill Cache subproblems begins to increase again after decreasing to 1.22 seconds for $P = 8$ and $k = 9$. For $P = 16$ and $P = 32$, the time spent on the small-sized Fill Cache subproblems also increases with the increase in the value of k. These two phenomena indicate that the small-sized Fill Cache subproblems cannot provide enough work for 16 and 32 processors. The small granularity of the parallel tasks impacts the performance negatively. Figure 4.4 suggests that the best solution is to solve these small-sized Fill Cache subproblems on only 8 processors.

As Figure 4.5 shows, the overall speedup for an alignment is close to the speedup for the Fill Cache subproblems, but the difference between the two speedups increases with the increase of P. This discrepancy increases mainly because the proportion of the Fill Cache time, out of the total time, decreases with the increase of P (Figure 4.3), and the impact the Fill Cache speedup has on the overall speedup also decreases. As expected, the speedup for the Base Case partitions is close to 1 (Figure 4.6) because the Base Case subproblems are solved sequentially. For up to 16 processors, the speedup for the initial Fill Cache subproblem is close to linear, whereas for 32 processors it goes as high as 19.43 for $k = 12$. The speedup for the initial Fill Cache subproblem can be seen as upper bound for the overall speedup of *Parallel FastLSA*.

Figure 4.7 shows a correlation between the increase in the percentage of the Fill Cache barrier time out of the total time and the increase of P. This can be seen more clearly in Figure 4.8, where the workload imbalance gets more accentuated with the increase of P. Also, the initial Fill Cache subproblems seem to suffer the most because of the larger granularity of their parallel tasks (i.e., tiles). As explained in Subsection 4.3.4, the barrier time is very likely to increase with the size of the tiles. The barrier times are irrelevant for the Base Case subproblems.

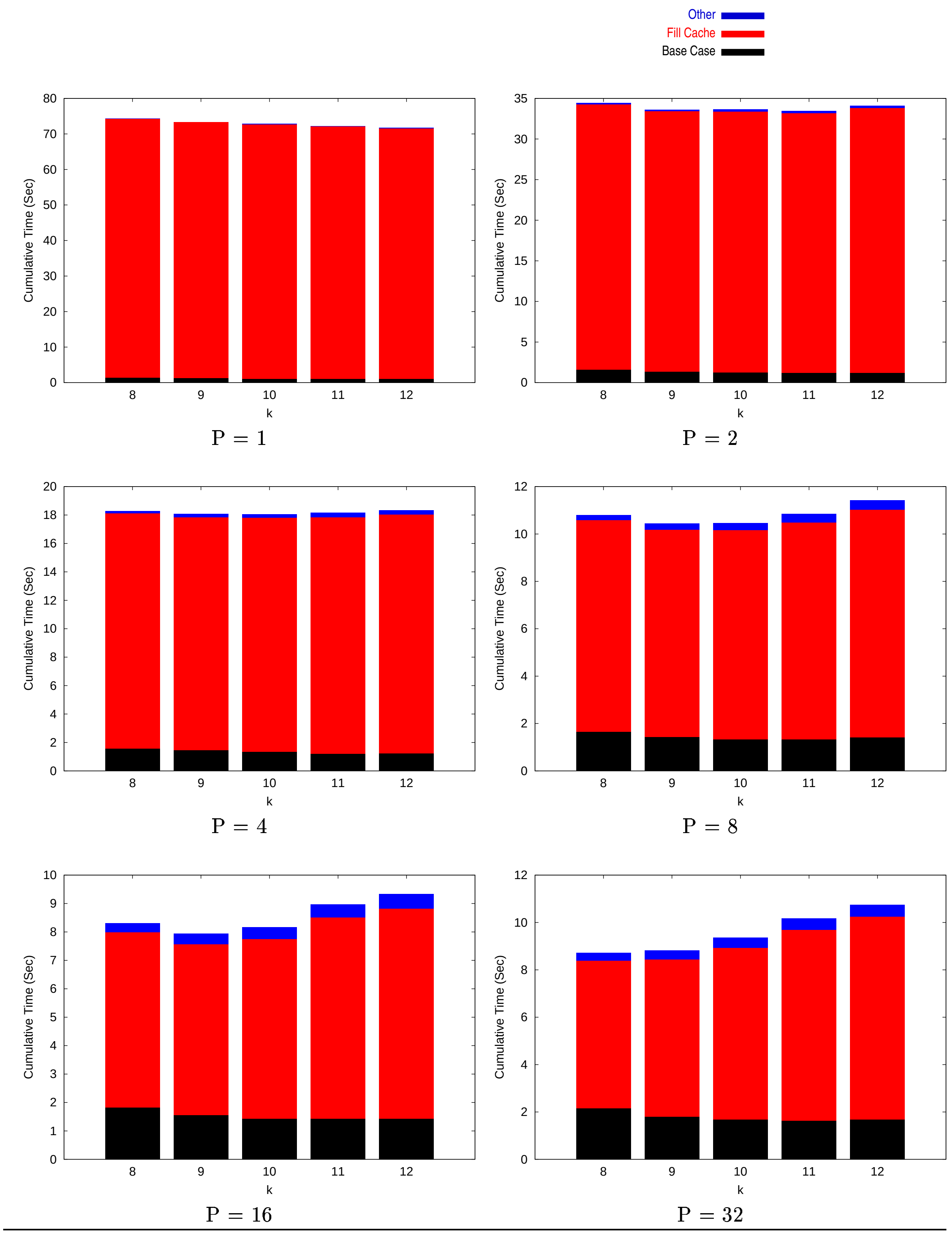
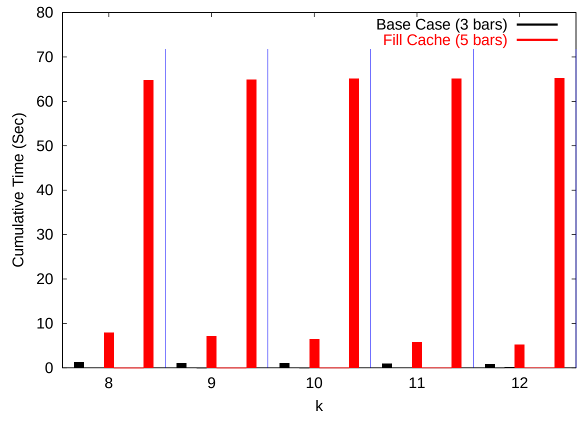
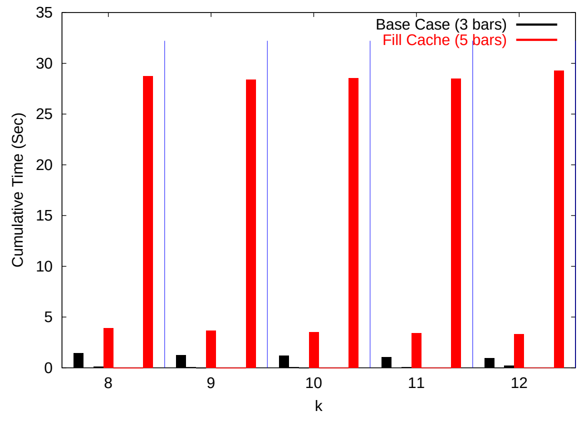


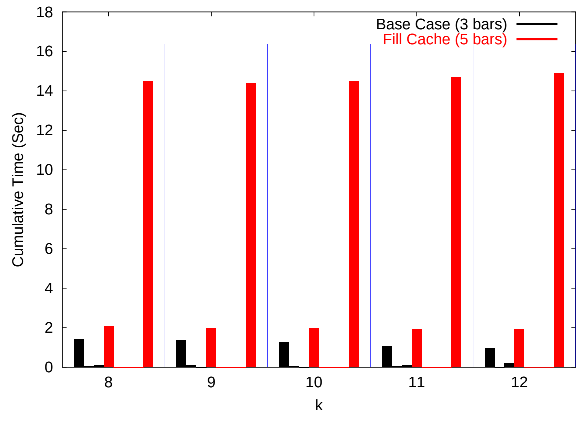
Figure 4.3: Execution Time: *Parallel FastLSA* Alignment for Human *XRCC1* versus Mouse *XRCC1* (Breakdown Based on the Type of the FastLSA Subproblems)



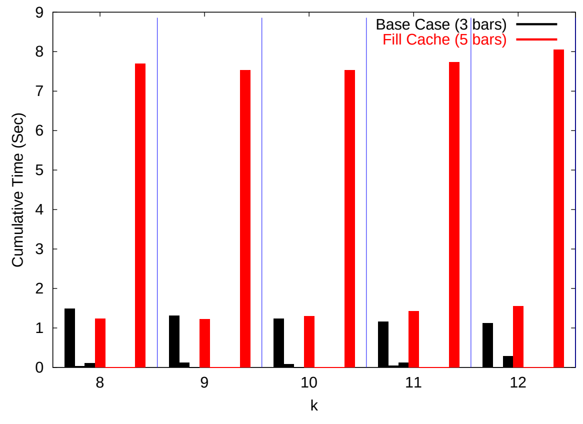
P = 1



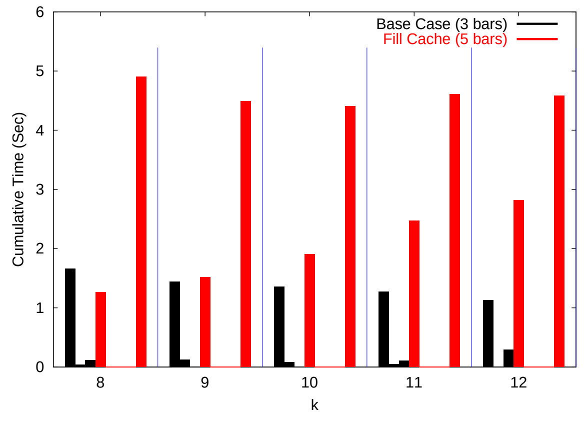
P = 2



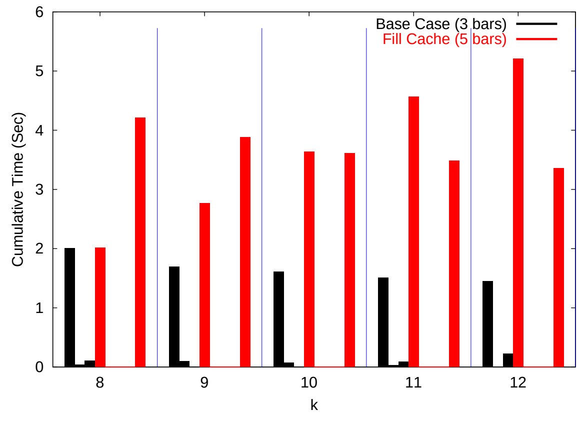
P = 4



P = 8



P = 16



P = 32

Figure 4.4: Execution Time: *Parallel FastLSA* Alignment for Human *XRCC1* versus Mouse *XRCC1* (Breakdown Based on the Size of the FastLSA Subproblems)

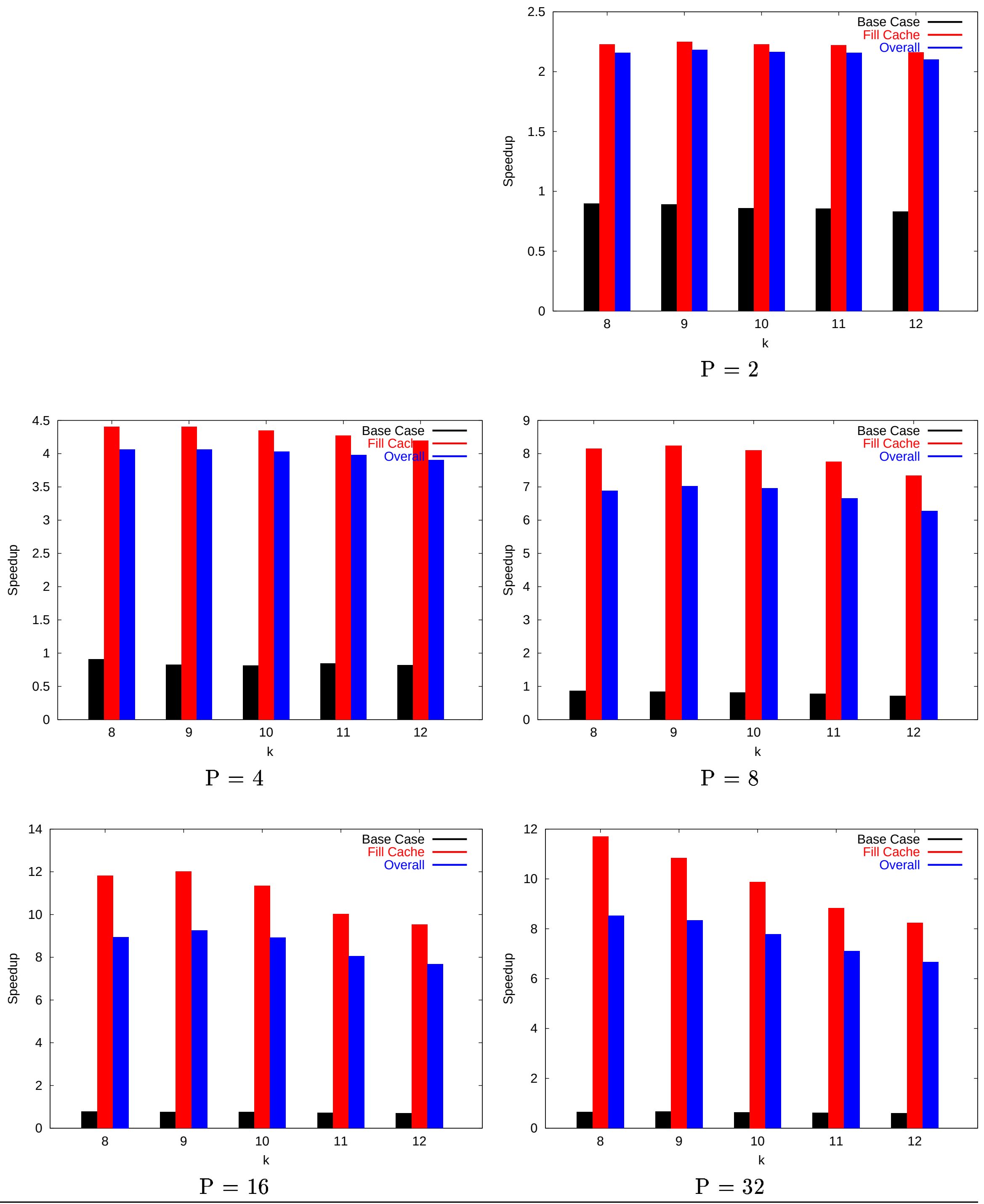


Figure 4.5: Speedup: *Parallel FastLSA* Alignment for Human *XRCC1* versus Mouse *XRCC1* (Breakdown Based on the Type of the FastLSA Subproblems)

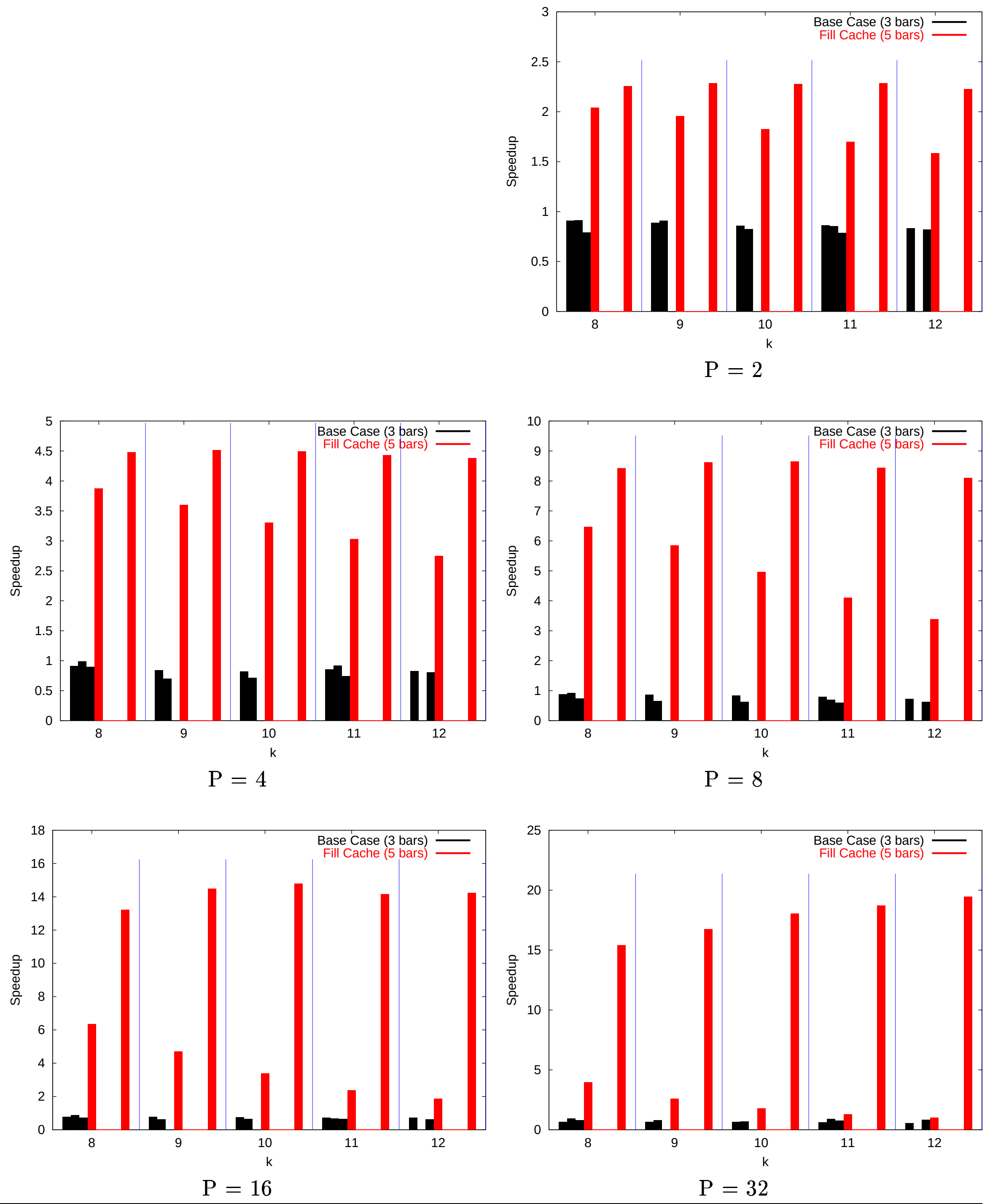


Figure 4.6: Speedup: *Parallel FastLSA* Alignment for Human *XRCC1* versus Mouse *XRCC1* (Breakdown Based on the Size of the FastLSA Subproblems)

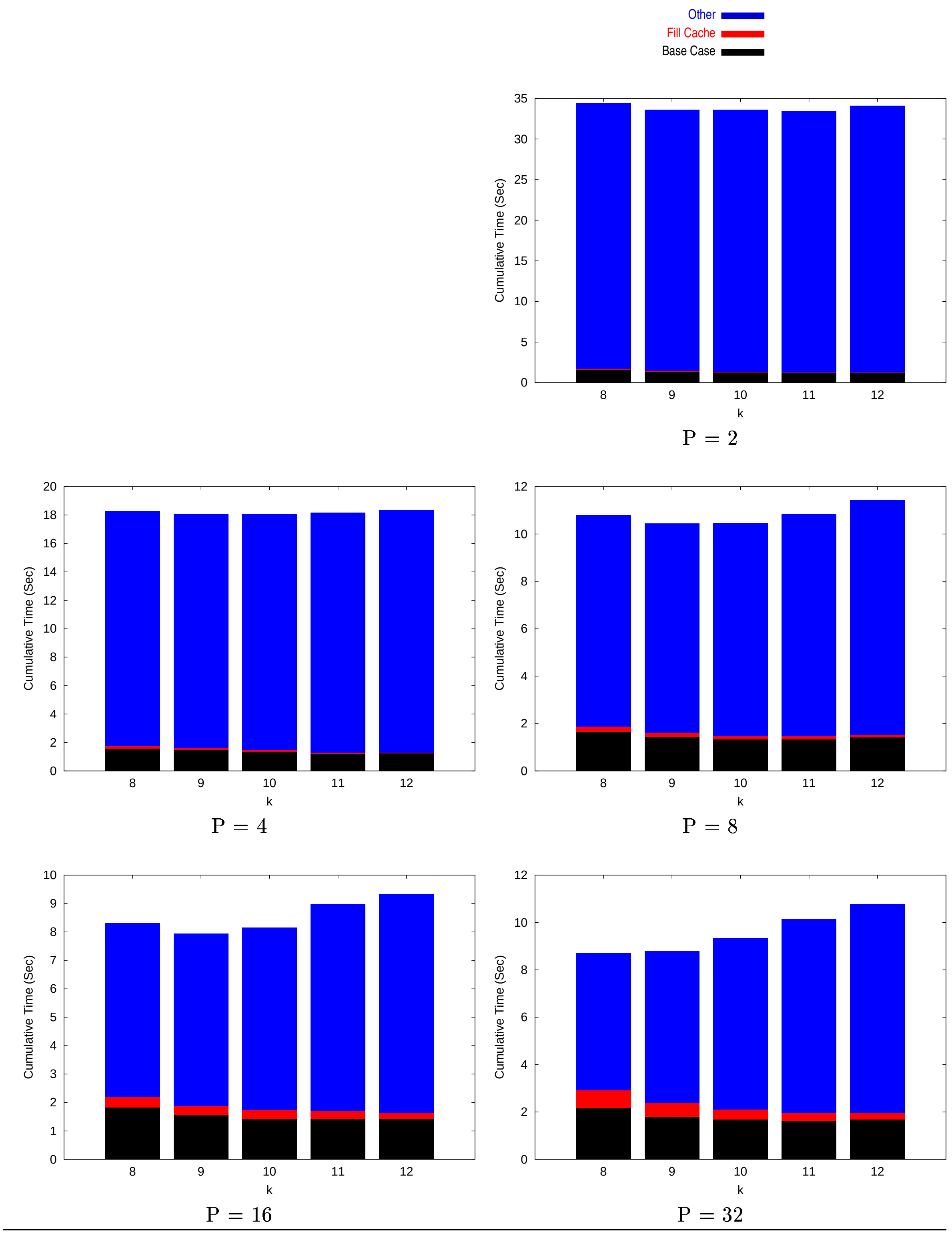


Figure 4.7: Barrier Time: *Parallel FastLSA* Alignment for Human *XRCC1* versus Mouse *XRCC1* (Breakdown Based on the Type of the FastLSA Subproblems)

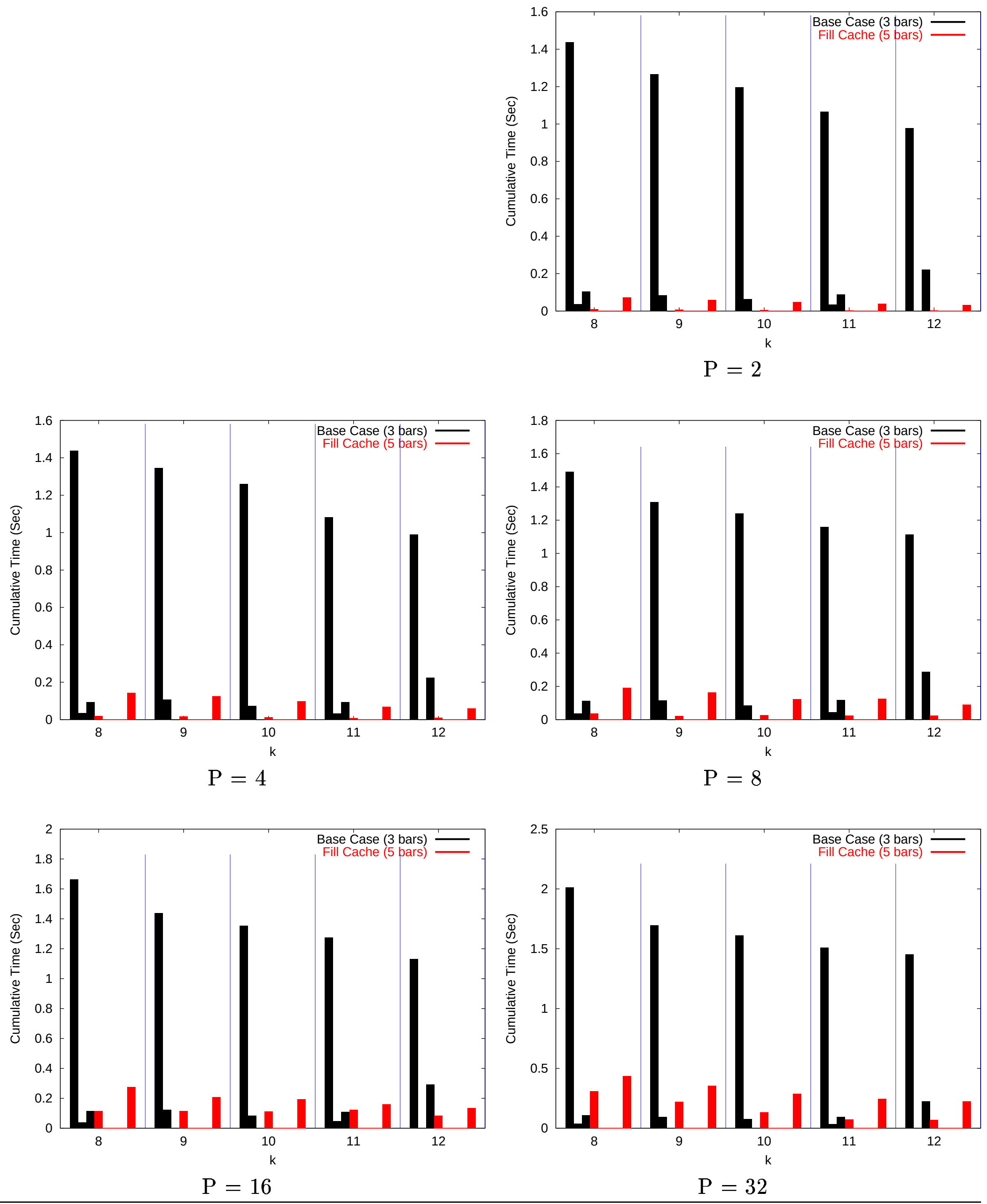


Figure 4.8: Barrier Time: *Parallel FastLSA* Alignment for Human *XRCC1* versus Mouse *XRCC1* (Breakdown Based on the Size of the FastLSA Subproblems)

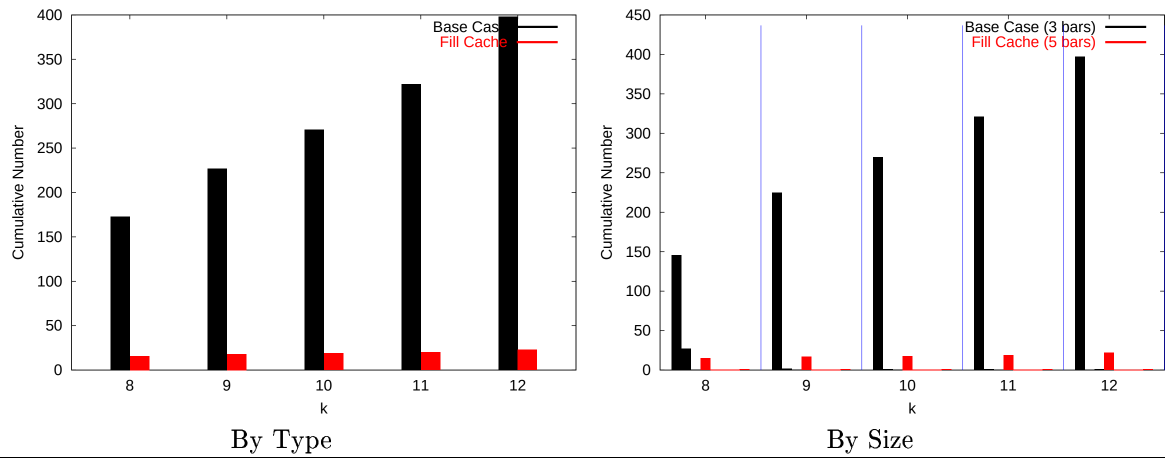


Figure 4.9: FastLSA Subproblem Count: *Parallel FastLSA* Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Type/Size of the FastLSA Subproblems)

4.5 *Myosin*

The subproblem count graph for the alignment of *Myosin* sequences (Figure 4.9) shows a noticeable increase in the number of Base Case subproblems from the previous data set (Figure 4.2), but only a modest increase in the number of Fill Cache subproblems. However, because of the increased size of the *Myosin* sequences, the Fill Cache subproblems are bigger, and better suited for parallelization than the Fill Cache subproblems for the *XRCC1* sequences.

Figure 4.10 exposes a trend in the values of the total execution time across the five values of k, which is similar to that observed for the *XRCC1* sequences: the optimal value of k shifts from 12 for $P = 1$ to 9 for $P = 32$. This phenomenon occurs because k controls the amount of re-computation that must be performed by the FastLSA algorithms and it also controls the granularity of the FastLSA subproblems and, indirectly, the granularity of the parallel tasks. A larger value for k means a larger FastLSA Grid Cache, a larger number of d.p. matrix entry values stored, and, therefore, less re-computation. When P has a small value, larger values for k tend to produce smaller execution times because less re-computation is performed than for smaller values of k. Because P is small, the contention for parallel work is small and the importance of the granularity of the parallel tasks is reduced. However, once P increases, the granularity of the parallel work becomes the dominant performance factor, overtaking re-computation time in importance. When P has a large value, the performance of *Parallel FastLSA* is best for small values of k because lower values for k tend to increase the granularity of the Fill Cache subproblems and, indirectly, the granularity of the parallel tasks.

Figure 4.11 shows an interesting trend for the time spent on the Fill Cache subproblems for 16 and 32 processors. While the time for the initial Fill Cache subproblem decreases with the increase of k, the cumulative time for the smallest Fill Cache subproblems decreases. Combining the information from the plots for $P = 16$ and $P = 32$, we can conclude that, for $P = 32$, a better execution time can be obtained by running the initial Fill Cache subproblem on 32 processors with $k = 12$, and the small Fill Cache subproblems on 16 processors with $k = 8$.

The speedup graph with type-based breakdown (Figure 4.12) shows a trend which is similar to that of the speedups for the *XRCC1* sequences: the overall speedup is close to the speedup for the Fill Cache subproblems, but the difference between the two speedups increases with the increase of P. Figure 4.13 provides useful information on the upper bounds for the overall speedups of *Parallel FastLSA* that can be expected when the parameter values listed in Table 4.1 are used for benchmarking. For example, when $P = 32$, the best speedup for the initial Fill Cache subproblem is 22.31, and it is obtained for $k = 12$. For 32 processors, 22.31 is a coarse upper bound for the best overall speedup, which was empirically determined to be 10.90 for $k = 9$.

The barrier time spent on solving the biggest Fill Cache subproblem (i.e., the initial FastLSA problem) decreases consistently with the increase in the value of k (Figure 4.15). It has already been mentioned that a higher k produces a higher number of FastLSA tiles, giving the opportunity for a better load balancing of work among the processors. Again, the percentage that the Fill Cache cumulative barrier time constitutes out of the total time increases dramatically with P (Figure 4.14). This is allowed by the fact that the upper bound for the barrier time is directly proportional with $P - 1$ (Subsection 4.3.4).

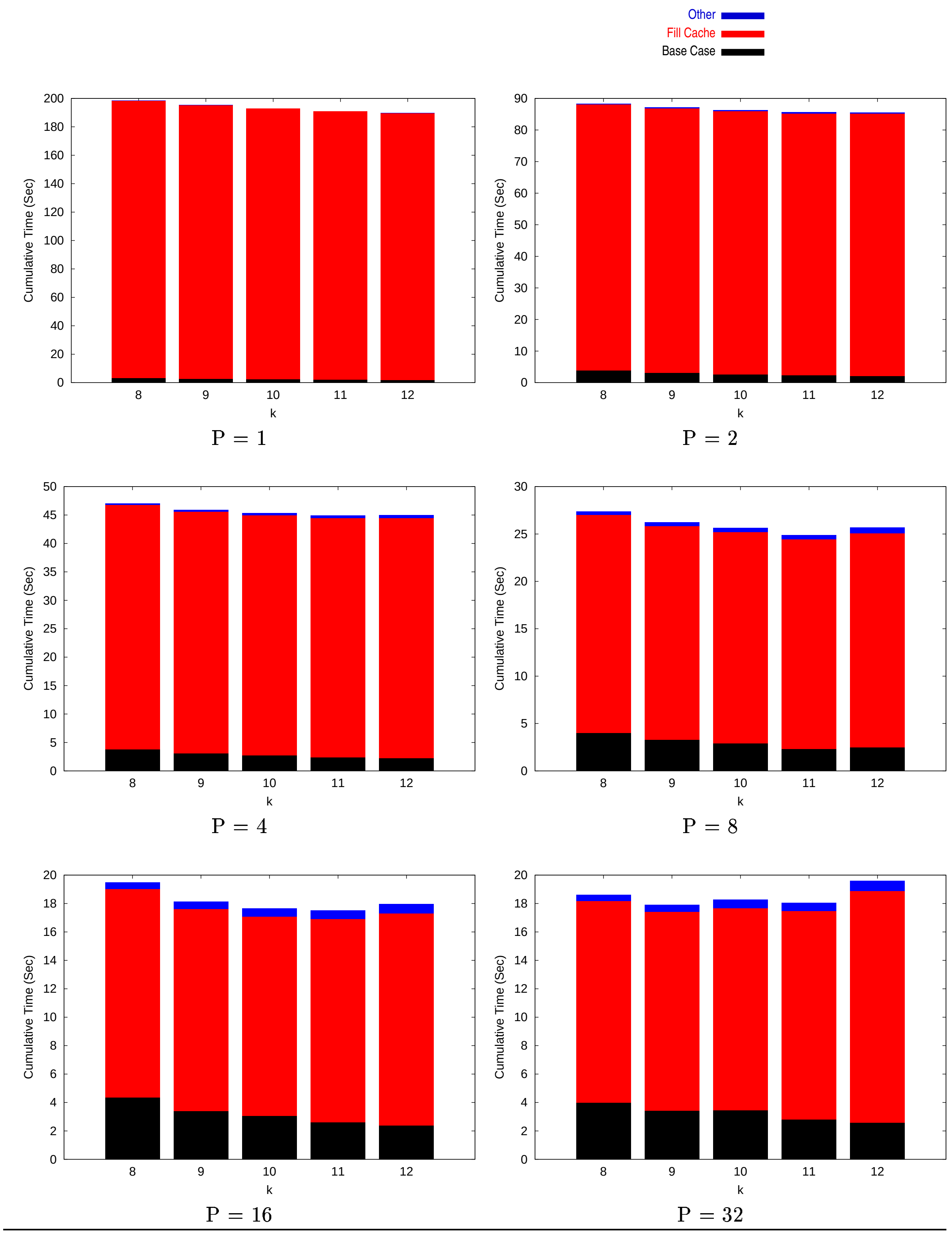


Figure 4.10: Execution Time: *Parallel FastLSA* Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Type of the FastLSA Subproblems)

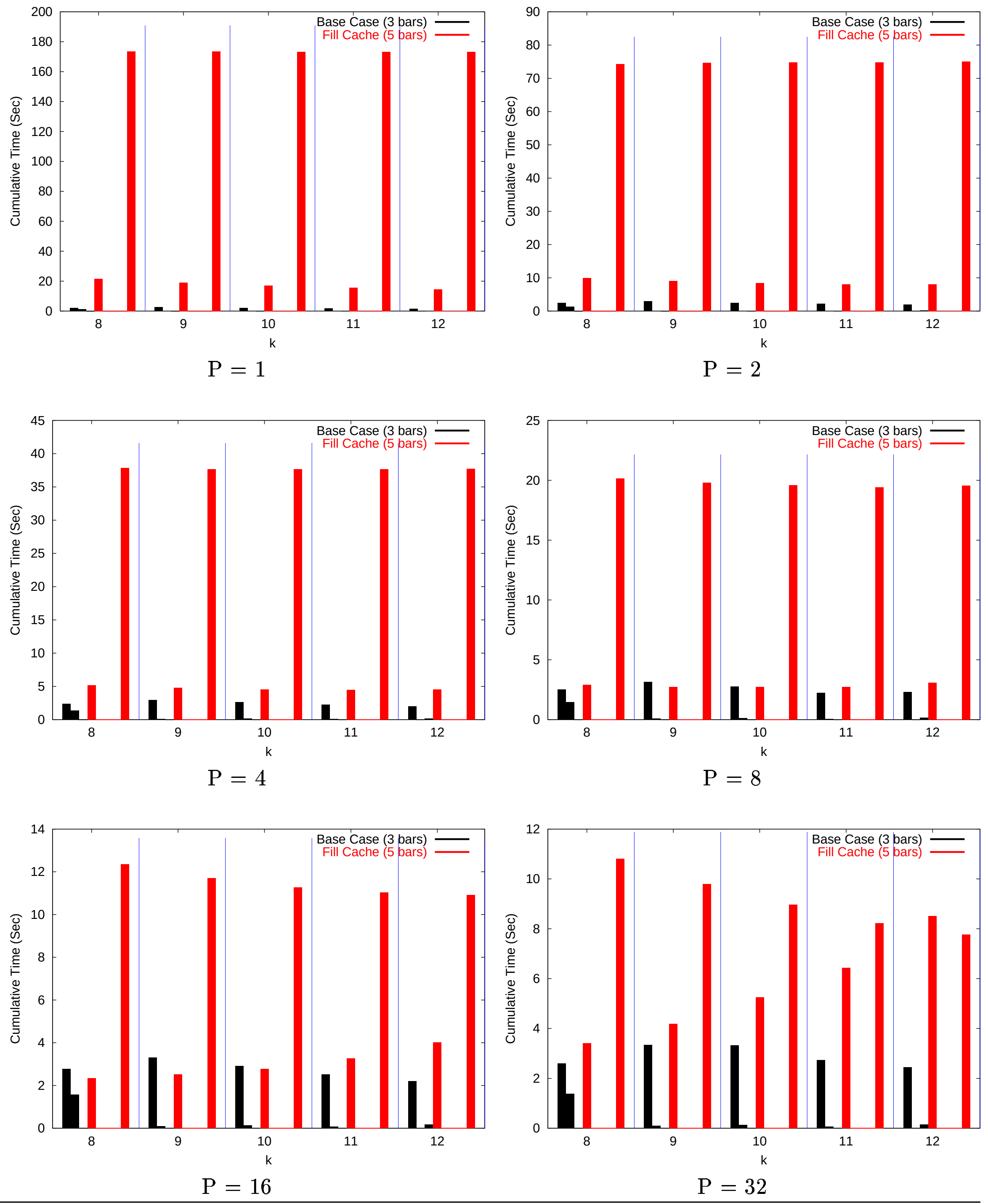
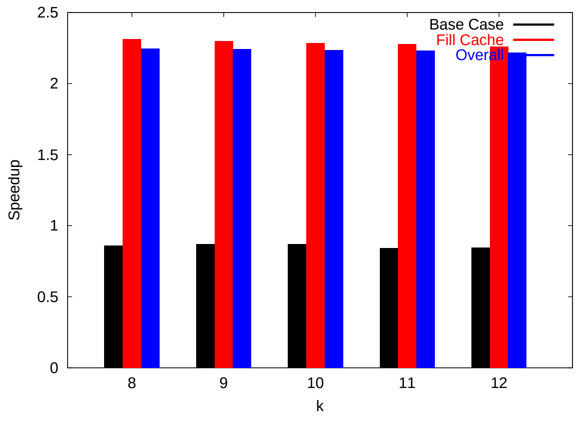
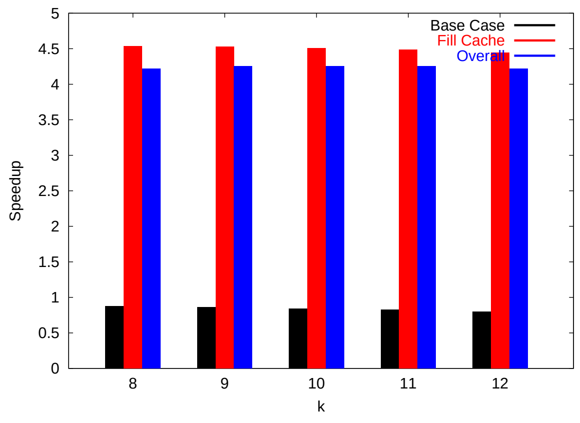


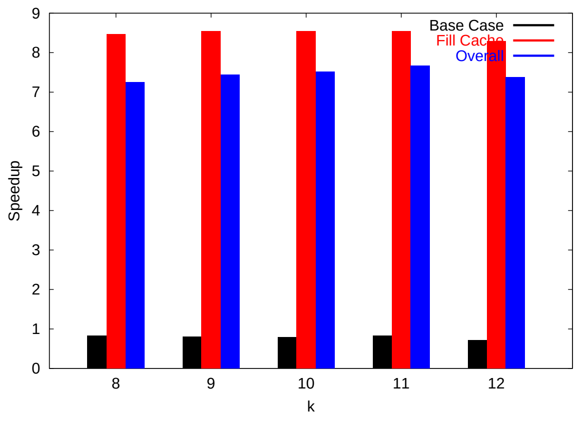
Figure 4.11: Execution Time: *Parallel FastLSA* Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Size of the FastLSA Subproblems)



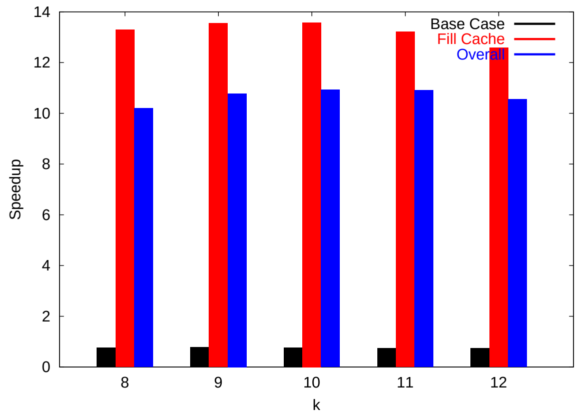
P = 2



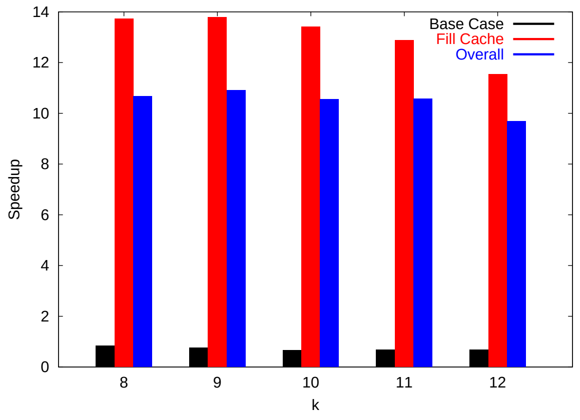
P = 4



P = 8



P = 16



P = 32

Figure 4.12: Speedup: *Parallel FastLSA* Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Type of the FastLSA Subproblems)

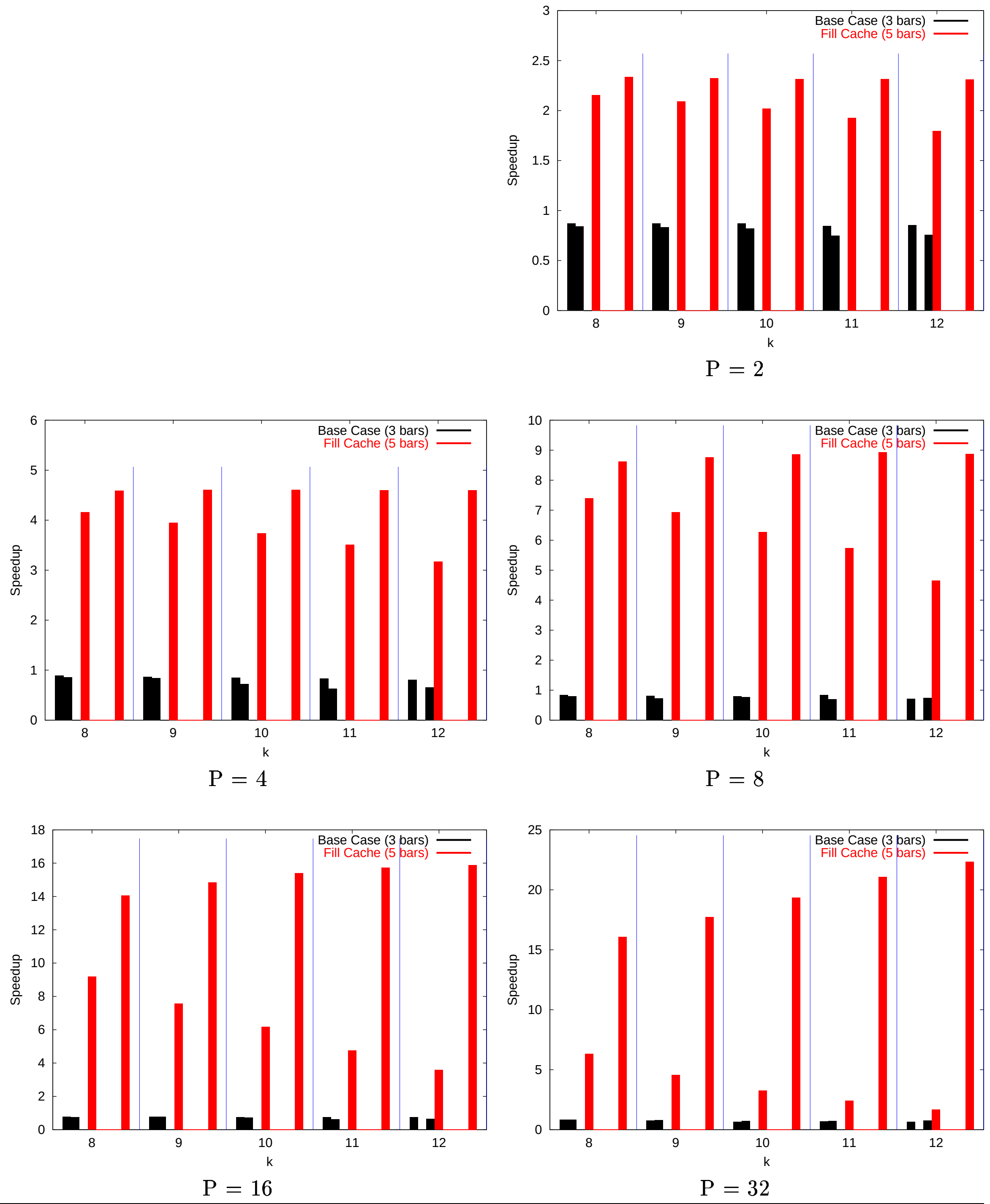


Figure 4.13: Speedup: *Parallel FastLSA* Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Size of the FastLSA Subproblems)

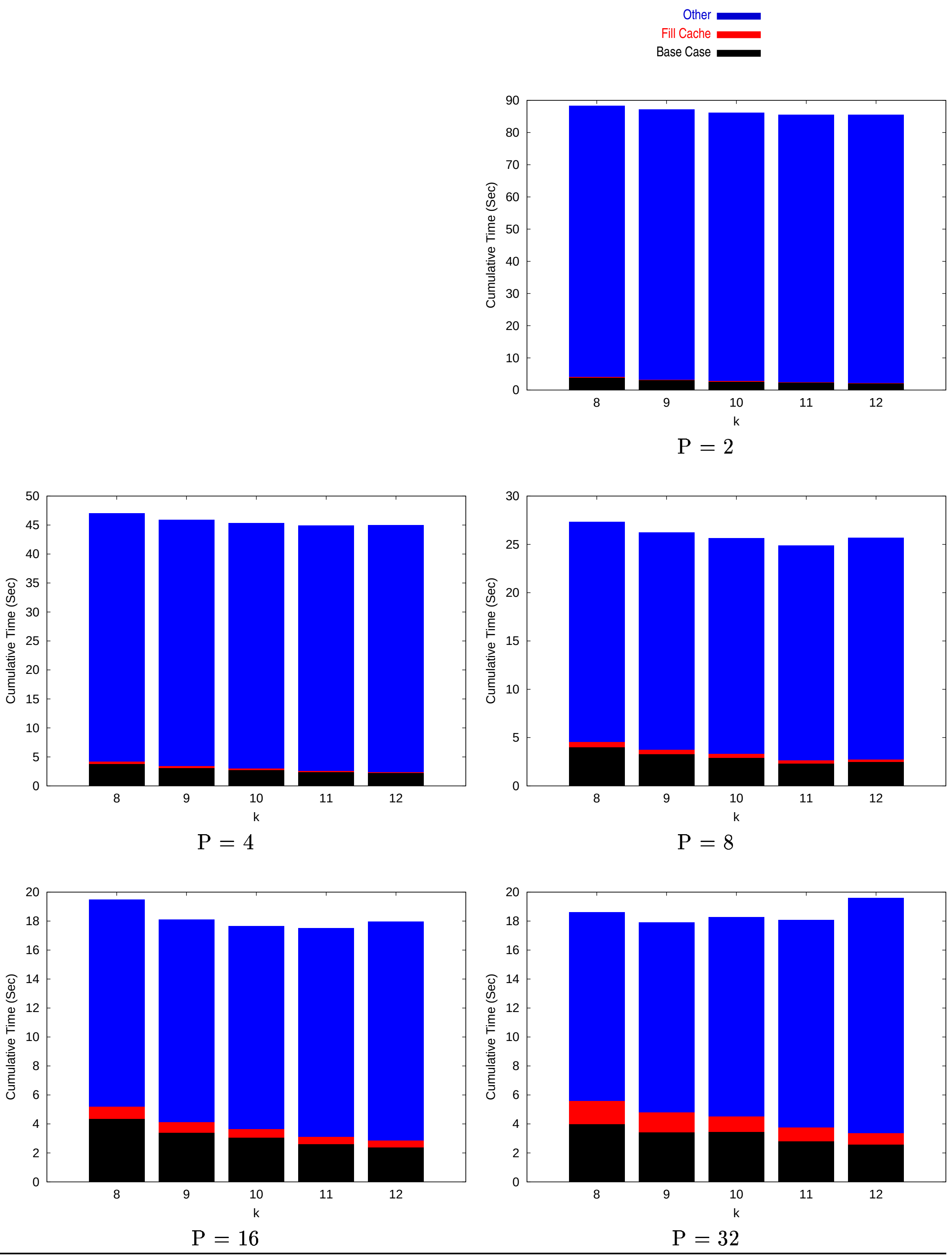


Figure 4.14: Barrier Time: *Parallel FastLSA* Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Type of the FastLSA Subproblems)

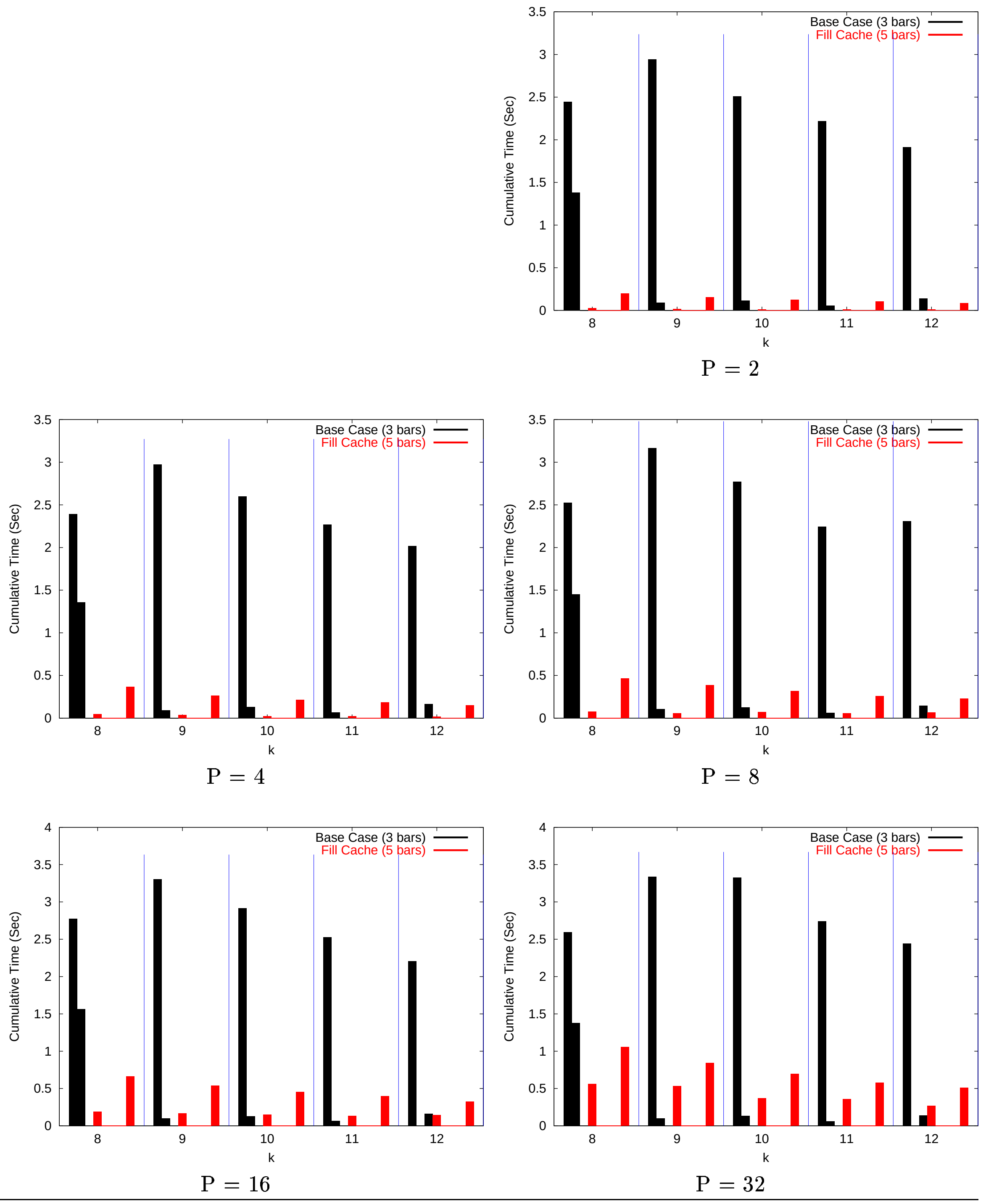


Figure 4.15: Barrier Time: *Parallel FastLSA* Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Size of the FastLSA Subproblems)

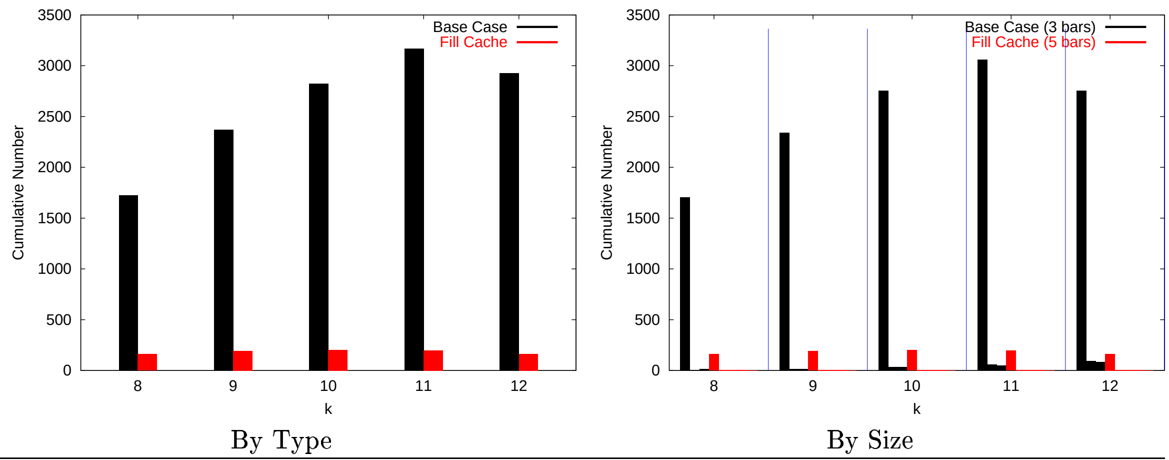


Figure 4.16: FastLSA Subproblem Count: *Parallel FastLSA* Alignment for Human *TCR* versus Mouse *TCR* (Breakdown Based on the Type/Size of the FastLSA Subproblems)

4.6 TCR

The subproblem count graph for the *TCR* sequences alignment (Figure 4.16) exhibits an interesting feature: the number of subproblems solved for $k = 12$ is considerably smaller than for $k = 11$. This defies the trend observed for the previous two pairs of sequences, for which the number of Base Case and Fill Cache subproblems increases almost linearly with k. The most likely explanation for this phenomenon is that, because of the configuration of the optimal path for the *TCR* sequences alignment and the configuration of the Grid Cache for $k = 12$, *Parallel FastLSA* avoids recomputing large parts of the d.p. matrix. A large number of subproblems are avoided because the *FastLSA* algorithm decides which subproblem to solve next based on the position of the current head of the optimal path in the d.p. matrix.

Figure 4.17 shows that, for every value of P, the optimal value for k is 12. This value is the best among the five values considered for k. Based on the trends observed for the previous two pairs of sequences, it can be concluded that higher values for k may work even better. We did not look for an empirically optimal value for k greater than 12, but plan to do so in our future work. The large size of the d.p. matrix for the alignment of the *TCR* sequences allows good granularity for the parallel tasks and, by increasing k, this can be traded off for a better load balancing of the parallel work. The optimal value of k is the one that generates the best equilibrium between granularity and load balancing.

It can be observed from Figure 4.18 that the best execution time for the alignment of the *TCR* sequences could be achieved by executing the initial Fill Cache subproblem on 32 processors, and all other Fill Cache subproblems on 16 processors, using $k = 12$ for all Fill Cache subproblems. For the Base Case subproblems, which are executed sequentially,

$k = 9$ is the optimal value among the values tested (Figure 4.17).

Figure 4.19 shows more clearly that the larger the sequences to be aligned, the more effective *Parallel FastLSA* is. For example, the best speedup for solving Fill Cache subproblems on 32 processors is 18.81 for the *TCR* sequences, while it is only 13.78 for *Myosin*, and 11.70 for *XRCC1*.

As for the previous pairs of sequences, Figure 4.20 helps to establish upper bounds for the overall speedup of *Parallel FastLSA*. For instance, the best speedup for the initial Fill Cache subproblem is 23.36 for 32 processors, 16.13 for 16, and 8.97 for 8 processors. These are much better speedups (i.e., linear for 16 processors or less) than those for *Myosin* or *XRCC1*.

The barrier times presented in Figure 4.21 show a trend consistent with that observed for the previous sequences: the percentage of the time spent at the barriers following the Fill Cache subproblems, out of the total time, increases with P. Figure 4.22 gives a good insight into how the value of k influences the distribution of the work on the processors. For every value of P considered, the cumulative barrier time for each partition set decreases with k. This occurs because a larger k generates more tiles, which in turn results in less processor starvation. The more parallel work is present in the queue, the less likely it is that the processors are blocked at the queue waiting for work. Also, because the tiles are smaller for a larger k, the final (i.e., third) phase of each parallel region is shortened, and the barrier time is reduced.

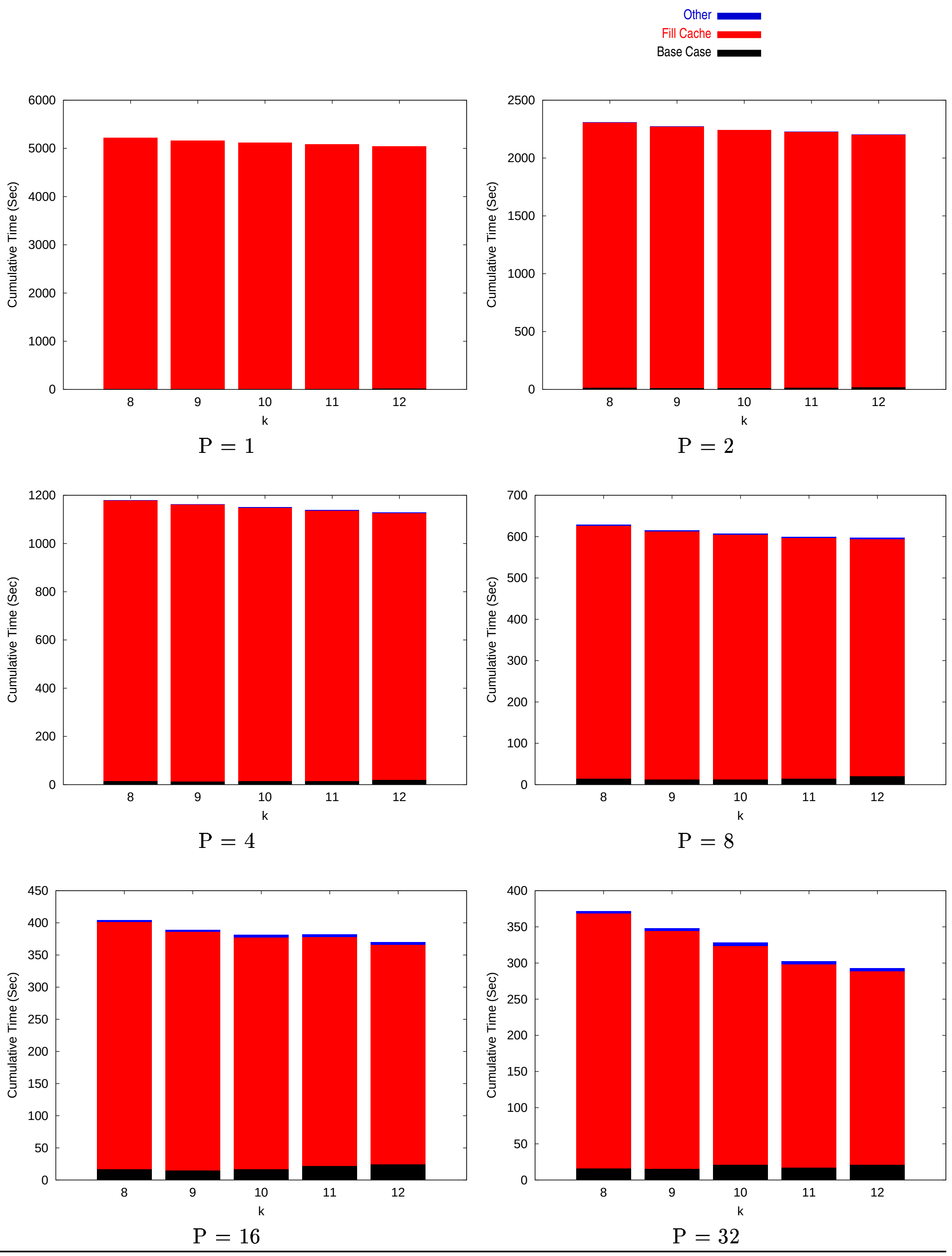


Figure 4.17: Execution Time: *Parallel FastLSA* Alignment for Human *TCR* versus Mouse *TCR* (Breakdown Based on the Type of the FastLSA Subproblems)

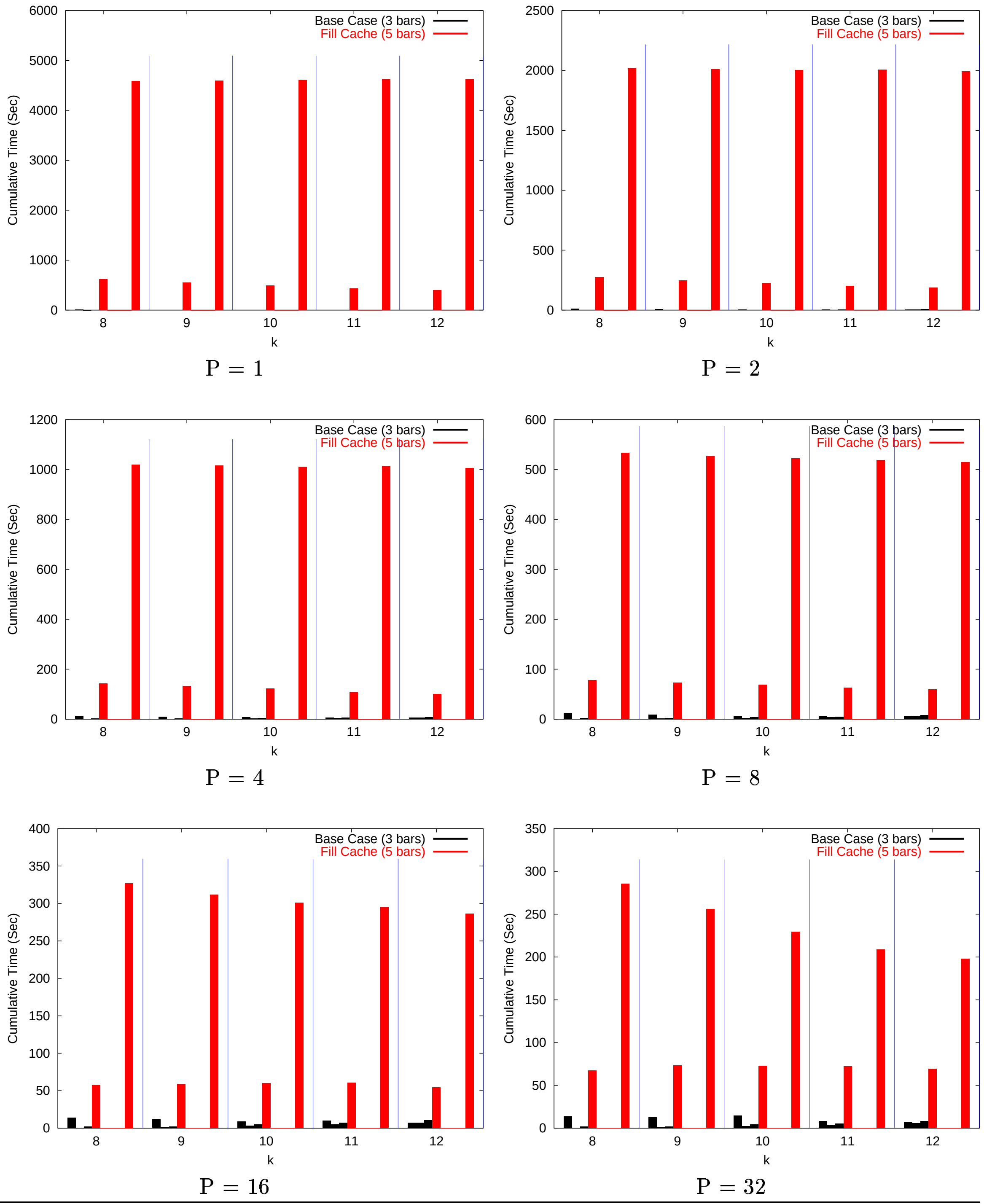


Figure 4.18: Execution Time: *Parallel FastLSA* Alignment for Human *TCR* versus Mouse *TCR* (Breakdown Based on the Size of the FastLSA Subproblems)

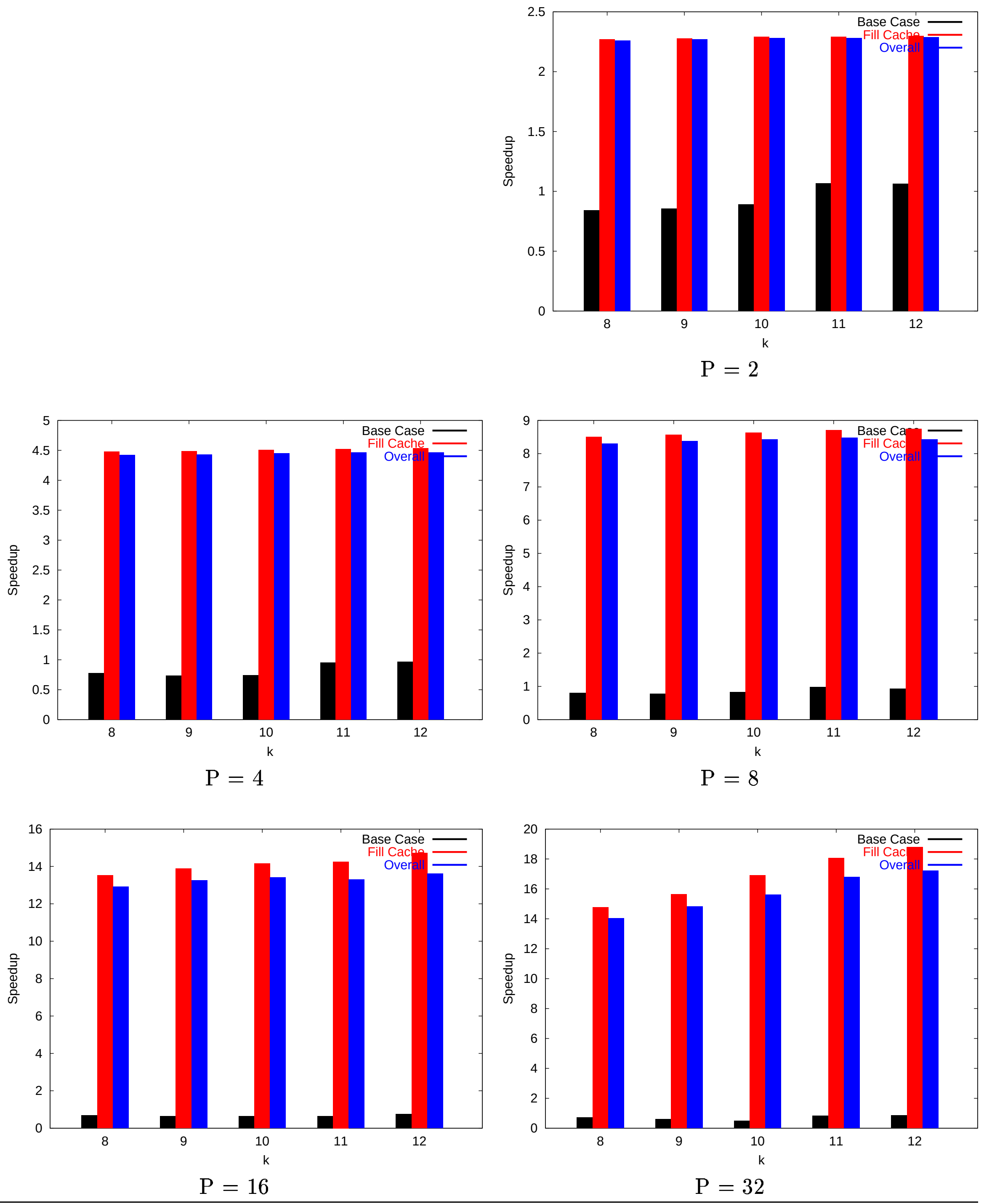


Figure 4.19: Speedup: *Parallel FastLSA* Alignment for Human *TCR* versus Mouse *TCR* (Breakdown Based on the Type of the FastLSA Subproblems)

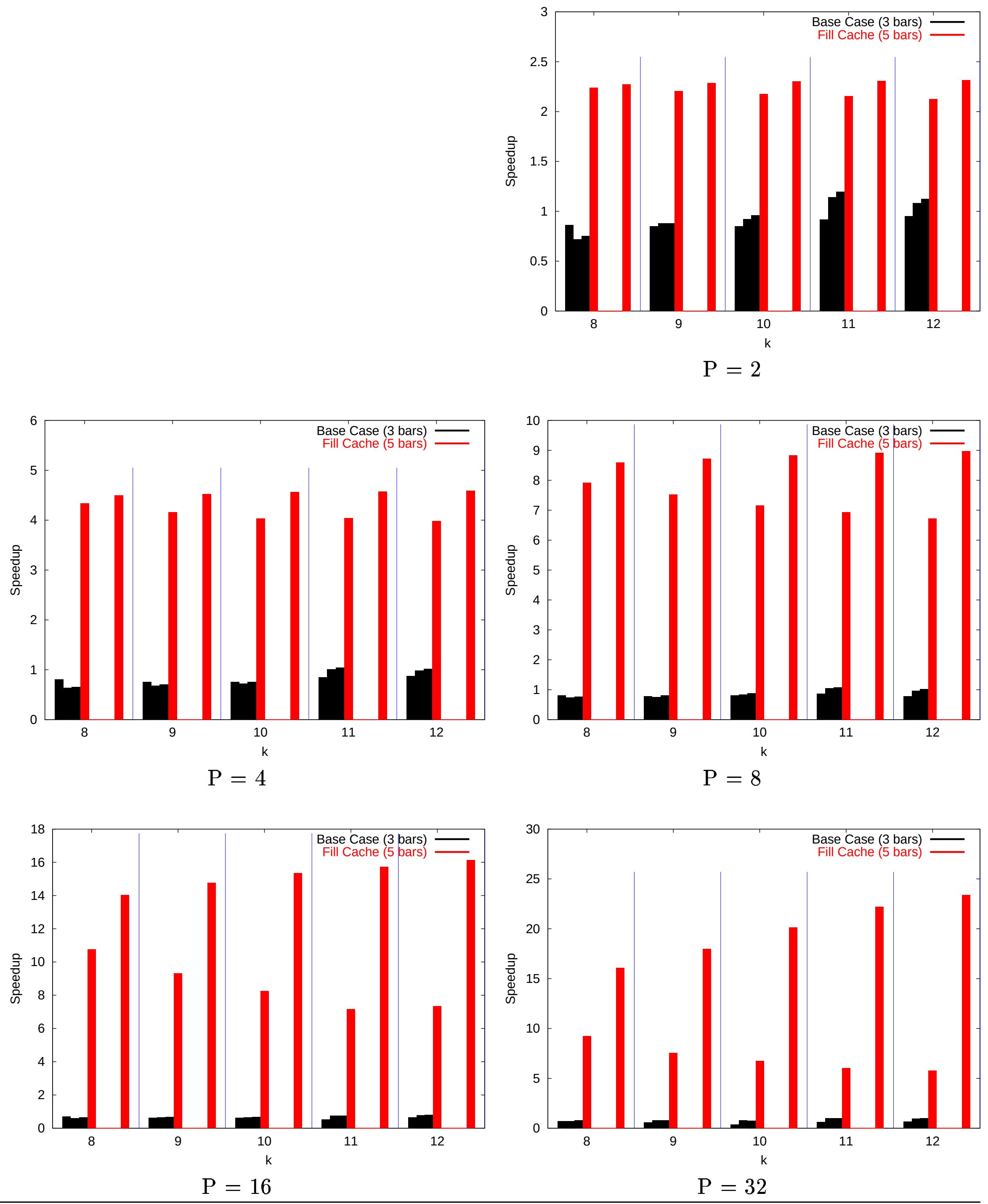


Figure 4.20: Speedup: *Parallel FastLSA* Alignment for Human *TCR* versus Mouse *TCR* (Breakdown Based on the Size of the FastLSA Subproblems)

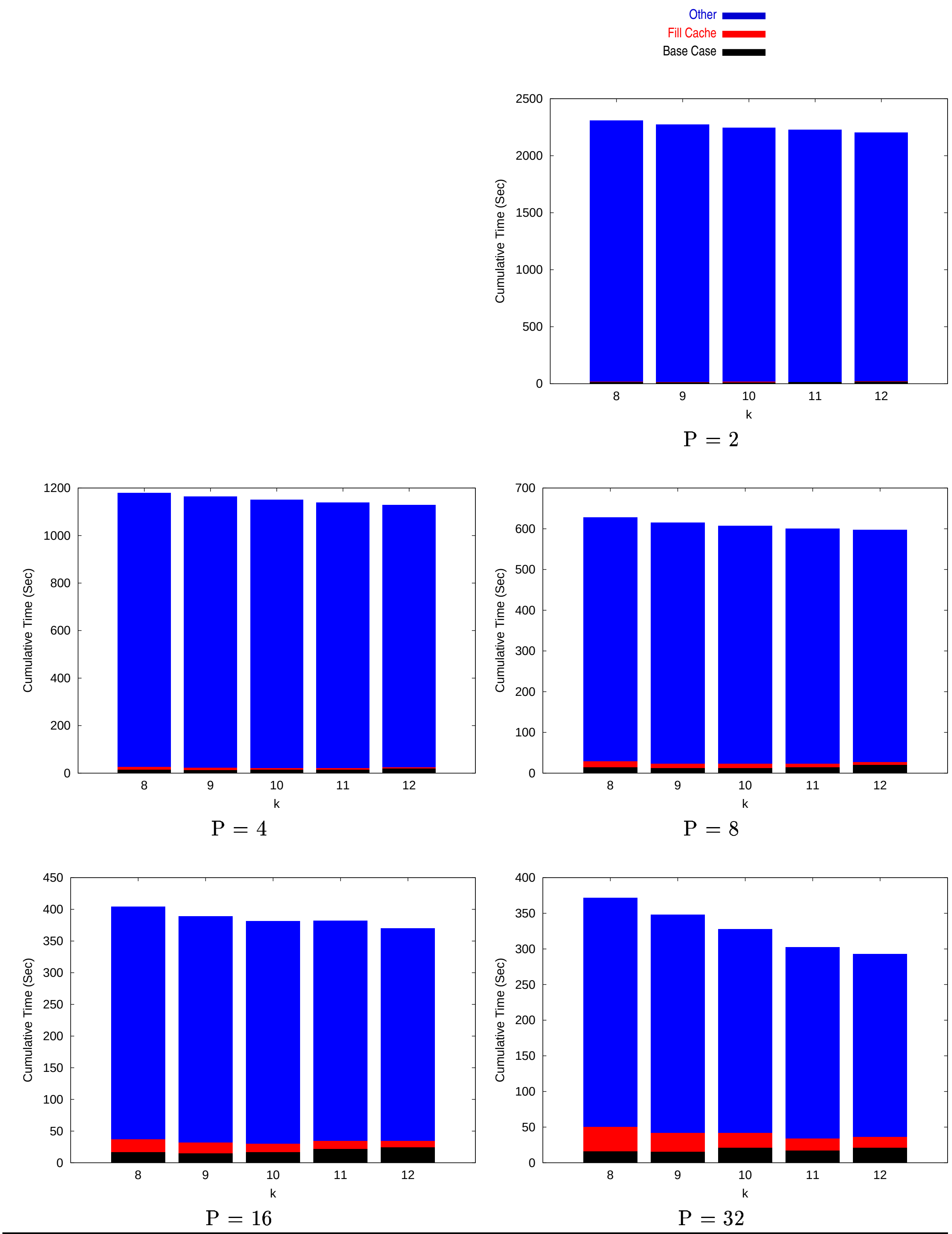
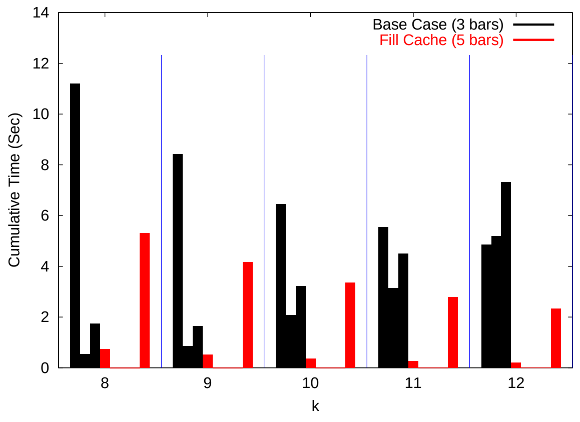
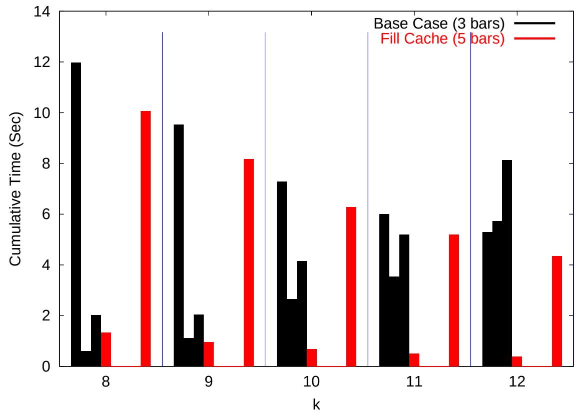


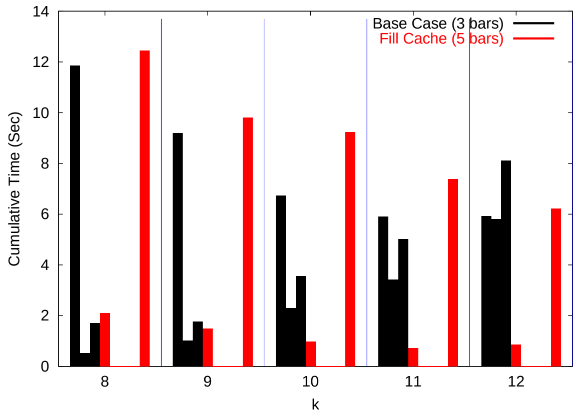
Figure 4.21: Barrier Time: *Parallel FastLSA* Alignment for Human *TCR* versus Mouse *TCR* (Breakdown Based on the Type of the FastLSA Subproblems)



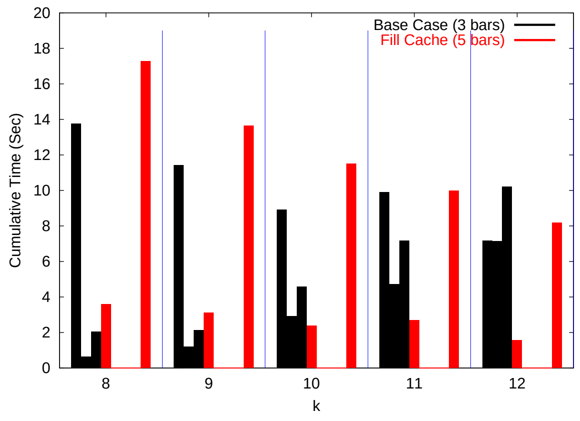
P = 2



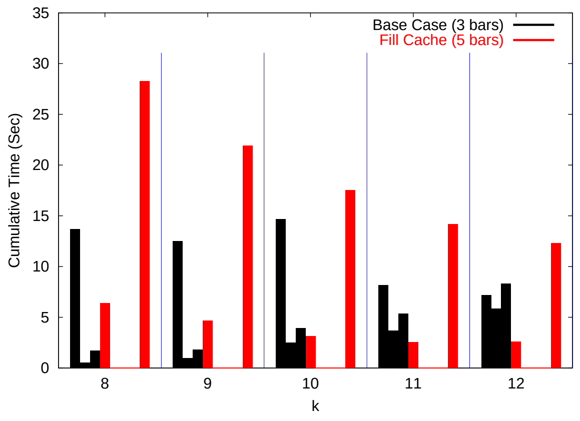
P = 4



P = 8



P = 16



P = 32

Figure 4.22: Barrier Time: *Parallel FastLSA* Alignment for Human *TCR* versus Mouse *TCR* (Breakdown Based on the Size of the FastLSA Subproblems)

4.7 Base Case Subproblems: Sequential Approach versus Parallel Approach

The *Parallel FastLSA* version that solves the Base Case subproblems sequentially was preferred to the version which solves the Base Case subproblems in parallel because it exhibits better performance. This is emphasized in Figure 4.23, which shows a pairwise comparison between the overall speedups for the two versions of *Parallel FastLSA*. The comparison is done for each pair of sequences and for each value of P.

When the Base Case subproblems are solved sequentially, the overall speedup is consistently better than when they are solved in parallel. The difference between the speedups for the two versions increases with P because of the poor performance of solving small Base Case subproblems on an increased number of processors. The poor performance is due to the large overhead associated with a large number of processors, and this overhead cannot be offset by the few opportunities for parallelism offered by the Base Case subproblems. Only a small number of very small tiles can be placed in the queue when a Base Case subproblem is solved in parallel and, consequently, only a few of the processors get to work on these tiles.

4.8 Concluding Remarks

This chapter describes the experiments performed with *Parallel FastLSA*, and discusses the results achieved. *Parallel FastLSA* exhibits good speedups, almost linear for 8 processors or less, and, not surprisingly, the longer the sequences to be aligned, the more efficient *Parallel FastLSA* is.

First, we present the experimental methodology used for our benchmarks. We specify the architecture on which *Parallel FastLSA* is run, and the three pairs of sequences that are aligned. We also list the values that are assigned to each parameter used by *Parallel FastLSA*. Some of these parameters are given several values in order to determine how their variation affects the performance of the algorithm.

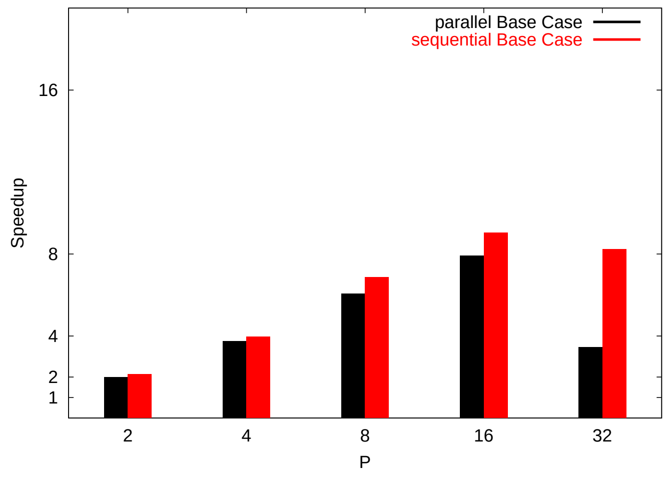
We then begin the discussion of the performance of *Parallel FastLSA* by displaying the best speedups obtained in our experiments. For 8 processors or less, the speedups are almost linear for all three sequence pairs that we align using *Parallel FastLSA*. We explain the trends of the speedup curves and hypothesize on the deterioration of the speedup that occurs from 16 to 32 processors.

In order to obtain a better insight into the effectiveness of *Parallel FastLSA*, we present for each pair of sequences a series of performance graphs. Each series consists of a subproblem count graph, execution time graphs, speedup time graphs, and barrier time graphs. These graphs are important because they help us assess the performance of *Parallel FastLSA* at the lower level of FastLSA subproblems.

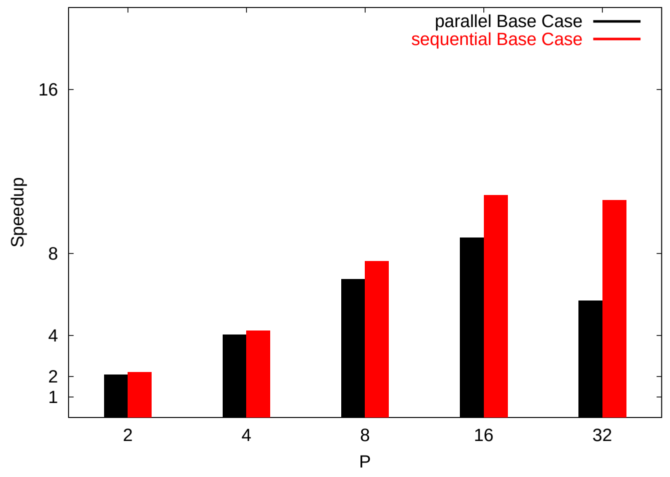
In the course of our experiments we discovered that running the Base Case subproblems in parallel causes a deterioration in the performance of *Parallel FastLSA*. For this reason, we mainly discuss experimental results for the version that runs the Base Case subproblems sequentially.

By studying the execution time graphs, we find out that an empirically optimal value for k can be found in the interval considered, $8 \leq k \leq 12$, but for the *TCR* sequences, for which an optimal value is most likely to be greater than 12. We also point out that the empirical optimal value for k tends to decrease when P increases.

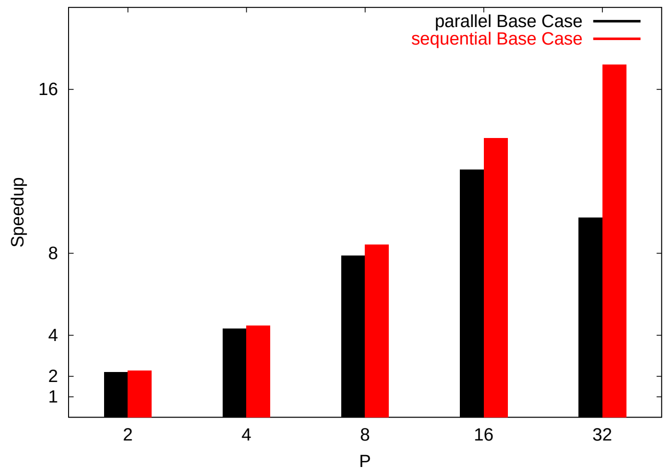
In the speedup graphs it can be observed that the overall speedup for an alignment is close to the speedup for the Fill Cache subproblems. Furthermore, we note that an almost linear speedup is achieved for the initial Fill Cache subproblem, which is the largest Fill Cache subproblem solved by *Parallel FastLSA* during an alignment operation.



XRCC1



Myosin



TCR

Figure 4.23: Comparison of the overall speedups for the two versions of *Parallel FastLSA*

Chapter 5

Future Work and Conclusions

5.1 Future Work

First, throughout our experiments in Chapter 4, we use constant, predetermined values for the parameters involved in *Parallel FastLSA*. Our goal for the future is to develop a procedure that will allow us to compute "good" values for BM, u, v, and k, based on the values of m, n, P, and the memory configuration of the computer on which *Parallel FastLSA* is run. These customized values for the parameters should allow *Parallel FastLSA* to align a pair of sequences in a time close to the empirical optimum.

Second, in the course of our research, we have decided that the FastLSA Grid Cache should have k rows and k columns for simplicity. However, there is no reason not to have a Grid Cache with k rows and l columns, where k and l can have different values. Furthermore, in the current implementation, we use the same value for k for every Fill Cache subproblem solved by the algorithm. Our belief is that it would be preferable to use customized values for k and l – values that are dynamically computed for each Fill Cache subproblem.

Third, our implementations of *Parallel FastLSA* are based on threads of execution, which are the trademark of the shared memory computational model. For this reason, we run all benchmarks on a shared memory multiprocessor computer. We have also considered implementing a distributed memory version of *Parallel FastLSA*, but deemed the communications penalty of distributing each tile across the network to be very time consuming. However, we wish to find out whether an MPI [Argonne National Laboratory, 2001] version of *Parallel FastLSA* may become efficient, and under what conditions.

5.2 Conclusions

Parallel FastLSA is a new parallel algorithm for optimal pairwise sequence alignment. *Parallel FastLSA* is the parallel version of *FastLSA*, a sequential algorithm that finds an optimal alignment for two biological sequences using linear space. *FastLSA* is proven to be empirically faster than two other frequently used algorithms: *Needleman–Wunsch* and *Hirschberg*. *Parallel FastLSA* is designed to further improve the time performance of *FastLSA*, while still using only linear space.

We have described the *Parallel FastLSA* algorithm in detail and have explained thoroughly how the algorithm can be implemented. In order to distribute parallel tasks to the processors efficiently, we propose two strategies: Dynamic Distribution of Work and Static Distribution of Work. These two strategies are implemented separately and are evaluated in our experiments. We conclude that their performance is similar, and discuss in detail only the results of the implementation based on Dynamic Distribution of Work.

We have analyzed the effectiveness of *Parallel FastLSA* and have given detailed accounts of its theoretical and empirical performance. The upper bounds that we find for the space and time complexity of *Parallel FastLSA* prove that *Parallel FastLSA* uses linear space and runs in quadratic time. Our experimental results show that *Parallel FastLSA* exhibits good speedups, almost linear for 8 processors or less, and its efficiency increases with the size of the sequences that are aligned.

Bibliography

Altschul, S. F., T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller and D. J. Lipman (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research* **25**, 3389–3402.

Altschul, S. F., W. Gish, W. Miller, E. W. Myers and D. J. Lipman (1990). Basic local alignment search tool. *Journal of Molecular Biology* **215**, 403–410.

Aluru, S., N. Futamura and K. Mehrotra (1999). Parallel biological sequence comparison using prefix computations. In: *Proceedings of the Second Merged Symposium IPPS/SPDP 1999, 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing.* San Juan, Puerto Rico. pp. 467–473.

Argonne National Laboratory, Mathematics and Computer Science Division (2001). MPICH. http://www-unix.mcs.anl.gov/mpi/mpich.

Campbell, M. K. (1999). *Biochemistry.* Saunders College Publishing, Harcourt Brace College Publishers.

Charter, K., A. Driga, P. Lu, J. Schaeffer, D. Szafron and I. Parsons (2002). FastLSA–a fast linear-space algorithm for sequence alignment. *in preparation.*

Charter, K., J. Schaeffer and D. Szafron (2000). Sequence alignment using FastLSA. In: *Proceedings of the 2000 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS 2000).* Las Vegas, Nevada. pp. 239–245.

Dayhoff, M., R. M. Schwartz and B. C. Orcutt (1978). A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure* **5**, 345–352.

Hirschberg, D. S. (1975). A linear space algorithm for computing longest common subsequences. *Communications of the ACM* **18**, 341–343.

Korf, R. E. (1999). A divide and conquer bidirectional search: First results. In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)* (Dean Thomas, Ed.). Morgan Kaufmann Publishers. San Francisco. pp. 1184–1191.

Korf, R. E. and W. Zhang (2000). Divide-and-conquer frontier search applied to optimal sequence alignment. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI-2000).* Austin, Texas. pp. 910–916.

Laudon, J. and D. E. Lenoski (1997). The SGI Origin: A ccNUMA Highly Scalable Server. In: *Proceedings of the 24th International Symposium on Computer Architecture.* Denver, Colorado. pp. 241–51.

Martins, W.S., J.B. del Cuvillo, F.J. Useche, K.B. Theobald and G.R. Gao (2001). A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In: *Pacific Symposium on Biocomputing 2001.*

Myers, E. and W. Miller (1988). Optimal alignments in linear space. *CABIOS* **4**, 11–17.

Needleman, S. B. and C. D. Wunsch (1970). A general method applicable to the search of similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* **48**, 443–453.

Penn State University (2001*a*). Bioinformatics Group. http://bio.cse.psu.edu.

Penn State University, Bioinformatics Group (2001*b*). TCR sequences. http://bio.cse.psu.edu/pipmaker/examples.html.

Penn State University, Bioinformatics Group (2001*c*). XRCC1 and Myosin sequences. http://globin.cse.psu.edu/globin/html/pip/examples.html.

Setubal, J. C. and J. Meidanis (1997). *Introduction to Computional Molecular Biology.* PWS Publishing Company.

Smith, T. F. and M. S. Waterman (1981). Identification of common molecular subsequences. *Journal of Molecular Biology* **147**, 195–197.

Veridian Systems (2001). PBS. http://www.pbspro.com.

Waterman, M. S. (1995). *Introduction to Computational Biology.* Chapman & Hall.