FastLSA: A Fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment

A. Driga[◊], P. Lu[◊], J. Schaeffer[◊], D. Szafron[◊], K. Charter, and I. Parsons

[◊]Department of Computing Science

University of Alberta

Edmonton, Alberta, T6G 2E8

Canada

{adrian|paullu|jonathan|duane}@cs.ualberta.ca

August 1, 2005

Running Head: Parallel and Sequential FastLSA

Contact Author:

```
Paul Lu
Associate Professor
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Canada

E-mail:        paullu@cs.ualberta.ca
Office:        (780) 492-7760
FAX:           (780) 492-1071
Web:           http://www.cs.ualberta.ca/˜paullu/
```

Abstract

Sequence alignment is a fundamental operation for homology search in bioinformatics. For two DNA or protein sequences of length m and n, full-matrix (FM), dynamic programming alignment algorithms such as Needleman-Wunsch and Smith-Waterman take O($m \times n$) time and use a possibly prohibitive O($m \times n$) space. Hirschberg's algorithm reduces the space requirements to O($min(m, n)$), but requires approximately twice the number of operations required by the FM algorithms.

The Fast Linear Space Alignment (FastLSA) algorithm adapts to the amount of space available by trading space for operations. FastLSA can effectively adapt to use either linear or quadratic space, depending on the specific machine. Our experiments show that, in practice, due to memory caching effects, FastLSA is always as fast or faster than Hirschberg and the FM algorithms.

To further improve the performance of FastLSA, we have parallelized it using a simple but effective form of wavefront parallelism. 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. Consequently, parallel and sequential FastLSA can be flexibly and effectively used with high performance in situations where space and the number of parallel processors can vary greatly.

Keywords: sequence alignment, homology search, bioinformatics, linear space, computational biology, parallel and sequential algorithms

1 Introduction

Sequence alignment is a fundamental operation in bioinformatics. Pairwise sequence alignment is used to determine homology (i.e., similar structure) in both DNA and protein sequences to gain insight into their purpose and function. Given the large DNA sequences (e.g., tens of thousands of bases) that some researchers wish to study [6, 19, 7], the space and time complexity of a sequence alignment algorithm become increasingly important.

As the first research contribution of this paper, we establish that the recently-introduced FastLSA [4] algorithm is the preferred sequential, dynamic programming algorithm for pairwise sequence alignment. Given FastLSA's strong analytical and empirical characteristics with respect to storage and time complexity, FastLSA is a good candidate for parallelization to improve its performance when dealing with large, whole genome alignments. As the second contribution, we show that FastLSA is nicely parallelizable while maintaining the strong space and time complexity properties of the sequential algorithm.

A recurring theme in this paper, and the third research contribution in the form of a case study, is the importance of algorithms (like FastLSA) that can be parameterized and tuned (e.g., via parameter k, discussed below) to take advantage of cache memory and main memory sizes. Existing algorithms for sequence alignment cannot be similarly parameterized. Furthermore, the selected value for parameter k has a significant impact on the parallel speedups of the algorithm, which results in interesting lessons in performance trade-offs.

1.1 Background

The primary structure of a protein consists of a sequence of amino acids, usually represented as a string, where each amino acid is represented by one of 20 different letters. To align two protein sequences, say `TLDKLLKD` and `TDVLKAD`, the sequences can be shifted right or left to align as many identical letters as

Symbol	Amino Acid Name	DNA Codon(s)	A	D	K	L	T	V
A	alanine	GC* (*=any)	16	-	-	-	-	-
D	aspartic acid	GAT GAC	0	20	-	-	-	-
K	lysine	AAA AAG	0	0	20	-	-	-
L	leusine	TTA TTG CT*	0	0	0	20	-	-
T	threonine	AC*	0	0	0	0	20	-
V	valine	GT*	0	0	0	12	0	20

Table 1: Part of Modified Dayhoff Scoring Matrix and Similarity Table, used for some examples in this paper

possible; in this example, 3 letters can be aligned (not shown). However, by allowing gaps ("-") to be inserted into sequences, we can often obtain more identical letters; in this example, there are 2 different ways of obtaining 5 identically aligned letters (highlighted by *):

```
TLDKLLK-D       TLDKLLK-D
T-DVL-KAD       T-D-VLKAD
* * * * *       * *  ** *
```

The different amino acids valine (V) and leucine (L) have similar functional properties so in sequence alignment we would like to indicate that the letters V and L are a better match than the amino acids lysine (K) and leucine (L), which have very different functional properties. To accommodate such similarity matches, we create a scoring function based on the numeric entries of a similarity table. For each pair of letters, the table gives a similarity score, where higher values indicate higher similarity. The score of an alignment is obtained by iterating over all pairs of corresponding letters in the aligned sequences and adding up the entries in the similarity table that is indexed by each pair. An optimal alignment is an alignment with the highest score for a given scoring function. In fact, there may be several optimal alignments with the same optimal score.

The similarity table for the scoring function used in this paper is based on the popular Dayhoff scoring matrix, MDM78 Mutation Data Matrix - 1978 [5]. It is the default similarity table used in the BioTools' commercial product PepTool (www.biotools.com). It has been scaled so that each entry is a nonnegative integer. Table 1 shows the part of the scoring table used in some of the examples of this paper. Higher scores denote higher similarity. Note that valine (V) and leucine (L) have a similarity score of 12, since they have similar function, while lysine (K) and leucine (L) have a similarity score of 0 to denote no similarity. If an amino acid in one sequence lines up with a gap in the other sequence, then a negative value, called a *gap penalty* is added to the score.

Many algorithms for sequence alignment are based on dynamic programming techniques that are equivalent to the algorithms proposed by Needleman and Wunsch [15] and Smith and Waterman [20]. Aligning two sequences of length m and n is equivalent to finding the maximum cost path through a dynamic program matrix (DPM) of size $m + 1$ by $n + 1$, where an extra row and column is added to capture leading gaps. Of course, high scores and the maximum cost paths are desirable with respect to the scoring functions in this paper. Given a DPM of size m by n, it takes O($m \times n$) time to compute the DPM cost entries, and then O($m + n$) time to identify the maximum cost path in the DPM. In this paper, algorithms that are based on storing the complete DPM are called full matrix algorithms (FM).

Unfortunately, calculations requiring O($m \times n$) space can be prohibitive. For instance, aligning two sequences with 10,000 letters each requires 400 Mbytes of memory, assuming each DPM entry is a single 4 byte integer. Although main memories in 2005 can be several hundred megabytes or gigabytes in size, the all-important processor caches are still (typically) well under 128 Mbytes. Furthermore, given that we now have the capacity to sequence entire genomes, pairwise sequence comparisons involving up to four million neucleotides at a time are now desirable. O($m \times n$) storage of this magnitude would require O(10^{13}) Mbytes of memory which is beyond the range of current technology.

Hirschberg [10] was the first to report a way of doing the computation using linear space. However, not storing the entire DPM means that some of the entries need to be recomputed to find the optimal path. It is a classic space-time trade-off: the number of operations approximately doubles, but the space overhead drops from quadratic to linear in the length of the sequences. In fact, Hirschberg's original algorithm was designed to compute the longest common sub-string of two strings, but Myers and Miller [14] applied it to sequence alignment.

In summary, there are two extremes for pairwise optimal sequence alignment:

1. full matrix, which minimizes the computational complexity, and

2. linear space, which minimizes the storage requirements.

However, linear-space alignment algorithms, such as Hirschberg's algorithm, do not take advantage of any additional memory that might be available.

This paper examines the FastLSA (Fast Linear-Space Alignment) algorithm, in both sequential and parallel versions. We expand on the original FastLSA paper [4] with new analytical and empirical results for the sequential algorithm. We also introduce a new parallel version of FastLSA [8] and provide substantial new analytical and empirical results. Compared to a previously-published version of this work [9], this paper provides the full proofs of the theorems (i.e., Appendix A), a more thorough coverage of the background and related work (i.e., this section and Section 2), and more empirical results (i.e., Section 4 and Section 6).

Unlike Hirschberg's algorithm, FastLSA can take advantage of extra space to reduce the number of operations. We describe the algorithms and we provide both analytical and empirical results for the algorithms. At one extreme, FastLSA uses linear space with approximately 1.5 times the number of operations required by the FM algorithms. At the other extreme, FastLSA uses quadratic space with no extra operations. Our experiments show that, in practice, due to memory caching effects, FastLSA is always as fast or faster than Hirschberg and the FM algorithms.

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. Consequently, parallel and sequential FastLSA can be flexibly and effectively used with high performance in situations where space and the number of parallel processors can vary greatly.

	-	T	L	D	K	L	L	K	D
-	0_{10}	-10	-20	-30	-40	-50	-60	-70	-80
T	-10	20_9	10_8	0	-10	-20	-30	-40	-50
D	-20	10	20	30_7	20_6	10	0	-10	-20
V	-30	0	22	20	30	32_5	22	12	2
L	-40	-10	20	22	20	50	52_4	42	32
K	-50	-20	10	20	42	40	50	72_3	62
A	-60	-30	0	10	32	42	40	62_2	72_A
D	-70	-40	-10	20	22	32	42	52_L	82_1

Figure 1: A Dynamic Programming Matrix (using similarity table from Table 1) and a Gap Penalty of -10. Subscripts denote an optimal path.

2 Related Work

2.1 Dynamic Programming and Full-Matrix Algorithms

FastLSA is a dynamic programming algorithm, like the FM algorithms and Hirschberg's algorithm, and it produces exactly the same optimal alignment for a given scoring function. The algorithms differ only in the space and time required.

The sequences from the introduction can be used to illustrate the differences between these algorithms. The scoring function uses the scoring table of Table 1 and a gap penalty of -10. Consider the sequences: TLDKLLKD and TDVLKAD. The alignment:

```
TLDKLLK-D
T-D-VLKAD
```

has an optimal score of (see Table 1, represented as SimilarityTable[]): SimilarityTable[T,T] + gap + SimilarityTable[D,D] + gap + SimilarityTable[L,V] + SimilarityTable[L,L] + SimilarityTable[K,K] + gap + SimilarityTable[D,D] = 20 + (-10) + 20 + (-10) + 12 + 20 + 20 + (-10) + 20 = 82. How is this optimal alignment obtained?

One sequence is placed along the top of the matrix and the other sequence is placed along the left side and a gap is added to the start of each sequence (Figure 1). Each different path from the top left corner to the bottom right corner of the matrix that goes only right, down or diagonal, represents a different alignment.

Any path can be translated to an alignment, but to obtain the optimal alignment for a given scoring function, we need to identify the corresponding optimal path. To derive the optimal path in the matrix, each of the three algorithms can be divided into two phases, which we call *FindScore* and *FindPath*. Figure 1 shows the DPM scores for the example sequences that are computed during the *FindScore* phase. The entries with numerical subscripts form the optimal path, that is computed in the *FindPath* phase.

In the *FindScore* phase, a 0 is placed in the upper-left corner of the matrix. Each algorithm propagates scores from the upper-left corner of the matrix to the lower-right corner. The score that ends up in the lower-right corner is the optimal score. The score of any entry is the maximum of the three scores that can be propagated from the entry on its left, the entry above it and the entry above-left. A diagonal move corresponds to a match or mismatch and adds the scoring table value for the two letters being considered. A down (right) move corresponds to inserting a gap in the horizontal (vertical) sequence and adds a gap penalty.

For example, the score of 20_9 in the ([T,T]) entry near the top left corner is the maximum of the scores from its left entry(-10 + -10 = -20), above entry (-10 + -10 = -20) and above-left entry (0 + SimilarityTable[T, T] = 0 + 20 = 20). The score of 10_8 in the ([T,L]) entry is the maximum of the scores from its left entry (20 + -10 = 10), its above entry (-20 + -10 = -30) and its above-left entry (-10 + SimilarityTable[T,L] = -10 + 0 = -10).

The FM algorithms, Hirschberg's algorithm and FastLSA all compute the score of the alignment in the same way. However, the FM algorithms store all of the $(m + 1) \times (n + 1)$ matrix entries, while the other two algorithms propagate a single row of scores (m entries) as the matrix is computed, overwriting an old row of scores by a new row of scores.

The *FindPath* algorithm computes the optimal path(s) backwards. For FM algorithms, the *FindPath* phase is straightforward. Since the FM algorithms store all scores in the DPM, they can compute the path by starting at the lower right corner and computing which of the three entries (left, up and diagonal) was used to compute its score. For example, the lower right ([D,D]]) entry is 82_1. Since its upper-left entry ([A,K]) has a score of 62_2 and since (62 + SimilarityTable[D,D] = 62 + 20 = 82), an optimal path goes through its upper-left ([A,K]) entry. In addition, an optimal path cannot lead to its above entry ([A,D]) with value 72_A since 72 - 10 = 62 $\neq$ 82. Similarly, an optimal path cannot lead to the left entry ([D,K]) whose value is 52_L. Note that in general it is possible for more than one path to be optimal. However, in our example, there is a single optimal path and it is denoted by numerical subscripts as shown in Figure 1. An alternative approach is to store three bits in each DPM entry to record the backward path. Each bit corresponds to one of the directions, diagonal, up or left. This will record multiple optimal paths. If only a single optimal path is required, two bits can be used to encode the three path choices at each DPM entry.

In the FM algorithms, the optimal path is easy to compute since the entire dynamic programming matrix is stored. However, neither Hirschberg's algorithm nor FastLSA stores the entire dynamic scoring matrix so the computation of the path is more complicated. In both cases, some of the DPM entries must be recomputed to find the path.

2.2 Hirschberg's Algorithm

Hirschberg's algorithm uses a divide-and-conquer approach. It splits one sequence in half (size $n/2$) and performs the *FindScore* computation on each half against the other original sequence (size m). However, the half-sequences are aligned from opposite ends or equivalently, the second half sequence is reversed. The algorithm does not store the entire DPM in memory. Instead only one row in each half matrix is stored and this row is updated as the computation continues. In essence, we are using a virtual or logical dynamic programming matrix without storing it.

After the two half alignments are complete, only the middle two rows of the matrix are known. This computation determines the split of the full sequence against the two half sequences. The split point maximizes the sum of the corresponding pairs of scores from the two half alignments.

Hirschberg's algorithm is called recursively to solve these two simpler problems. The size of the sub-problem is $n/2$ by approximately $m/2$, depending on where the split occurred. Since the DPM is not stored, parts of it will need to be re-computed.

The recursion terminates when the size of the sub-problems is one, but it could be terminated sooner by using a FM algorithm when the problem size is small enough to solve in memory or cache. Approximately $m \times n$ re-computations need to be done using Hirschberg's algorithm [14].

2.3 Parallel Dynamic Programming

In the broader area of the design and analysis of parallel algorithms, dynamic programming has been studied by many of researchers. The spectrum of papers ranges from the theoretical (e.g., [3, 1]) to papers with applied and empirical results, in addition to theory (e.g., [12, 13]).

Dynamic programming solves a large number of diverse applications, ranging from, for example, string edit distance [1] to sequence alignment [13] (i.e., the motivation for FastLSA itself). There are differences in the allowed operations (e.g., deletion, insertion, and substitution in string editing as opposed to matching, mismatching, and inserting a gap in sequence alignment). But, there are also similarities between applications at the level of the dynamic programming paradigm: the results of partial subproblems are combined to solve larger problems. Consequently, the concepts of pipelining and dependencies between subproblems (e.g., [12]) and the strategies for combining the results of subproblems (e.g., [1]) are re-visited by different researchers. Furthermore, for each application, there can be different assumptions about the granularity of the tasks (e.g., modelled as a random variable following a probability distribution [12]) and about the common problem sizes (e.g., sequence lengths less than 1,000 characters [12] versus tens or hundreds of thousands of characters (Table 3)).

Given the large spectrum of possible analyses, applications, and assumptions, direct comparisons between results are difficult. However, to provide some context, our work with FastLSA is more towards the applied and empirical end of the spectrum. The development of FastLSA was driven by the desire to improve sequence alignment performance in practice. Our empirical results come from problem sizes taken from actual biological data (Table 3) and an implementation running on contemporary hardware (Section 6). We used our analytical results to better understand the implementation issues related to the algorithm. For example, the trade-off between time and space lends itself to theoretical analysis, but the empirical analysis is the ultimate validation of this principle.

3 Sequential FastLSA Algorithm

We describe the FastLSA algorithm and show how it is different from both the FM and Hirschberg algorithms. In particular, FastLSA can be tuned to take advantage of different cache memory and main memory sizes. Furthermore, we show that FastLSA is the preferred algorithm in practice, which also makes it a good candidate for parallelization.

The basic idea of FastLSA [4, 8] is to use more available memory to reduce the number of re-computations that need to be done in Hirschberg's algorithm. This is accomplished by: (1) dividing both sequences instead of just one, (2) dividing each sequence into k parts instead of only two and (3) storing some specific rows and columns of the logical dynamic programming matrix (DPM) in grid lines to reduce the re-computations.

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. RM may represent either the size of cache memory or main memory, depending on the specific performance-tuning goal of the programmer. If $RM > m \times n$, then a full matrix algorithm (e.g., Needleman-Wunsch) can be

```
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 3.6 (a) */
1   if flsaProblem fits in allocated buffer then
        // BASE CASE
        /* Figure 3.6 (b) */
2       return solveFullMatrix( flsaProblem, cacheRow, cacheColumn, flsaPath )

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

    /* Figure 3.6 (c) */
5   fillGridCache( flsaProblem, flsaGrid )

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

    /* Figure 3.6 (d) */
8   flsaPathExt = FastLSA( 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 3.6 (e) */
13      flsaPathExt = FastLSA( flsaSubProblem, newCacheRow, newCacheColumn, flsaPathExt )

14  deallocateGrid( flsaGrid )

    /* Figure 3.6 (f) */
15  return flsaPathExt
```

Figure 2: Pseudo-Code for FastLSA

used to solve the problem because the DPM can be stored in the available memory.

FastLSA is a recursive algorithm based on the divide-and-conquer paradigm. The pseudo-code for the FastLSA algorithm is shown in Figure 2. A call to FastLSA takes as input a logical DPM corresponding to a pair of sequences and an optimal solution path that ends at the bottom-right entry of this logical DPM. 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 DPM entry values are also passed in with each call to FastLSA.

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 DPM to form a complete optimal path.

For the initial call to FastLSA, the logical DPM 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 DPM has $(m+1) \times (n+1)$ entries whose values must be computed. The initial optimal path, *flsaInitialPath*, is formed from a single point, (m, n), the bottom-right entry of the original logical DPM.

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 DPM 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).

The full matrix algorithm uses the input values *cacheRow* and *cacheColumn* as the first row and column of the DPM it must compute (Figure 3(a)). After all entries of the DPM have been computed, an optimal path through the matrix is built. Figure 3(b) shows the computed and stored DPM 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 DPM for the input problem is larger than BM, the General Case of the algorithm is followed (line 3 onwards in Figure 2). 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 DPM into k equal segments, $k \geq 2$. As a result, the DPM for the input problem is partitioned into k^2 *logical sub-matrices* of size approximately $\frac{m}{k} \times \frac{n}{k}$ (Figure 3(c)). These sub-matrices are laid out in k rows, each row having k columns.

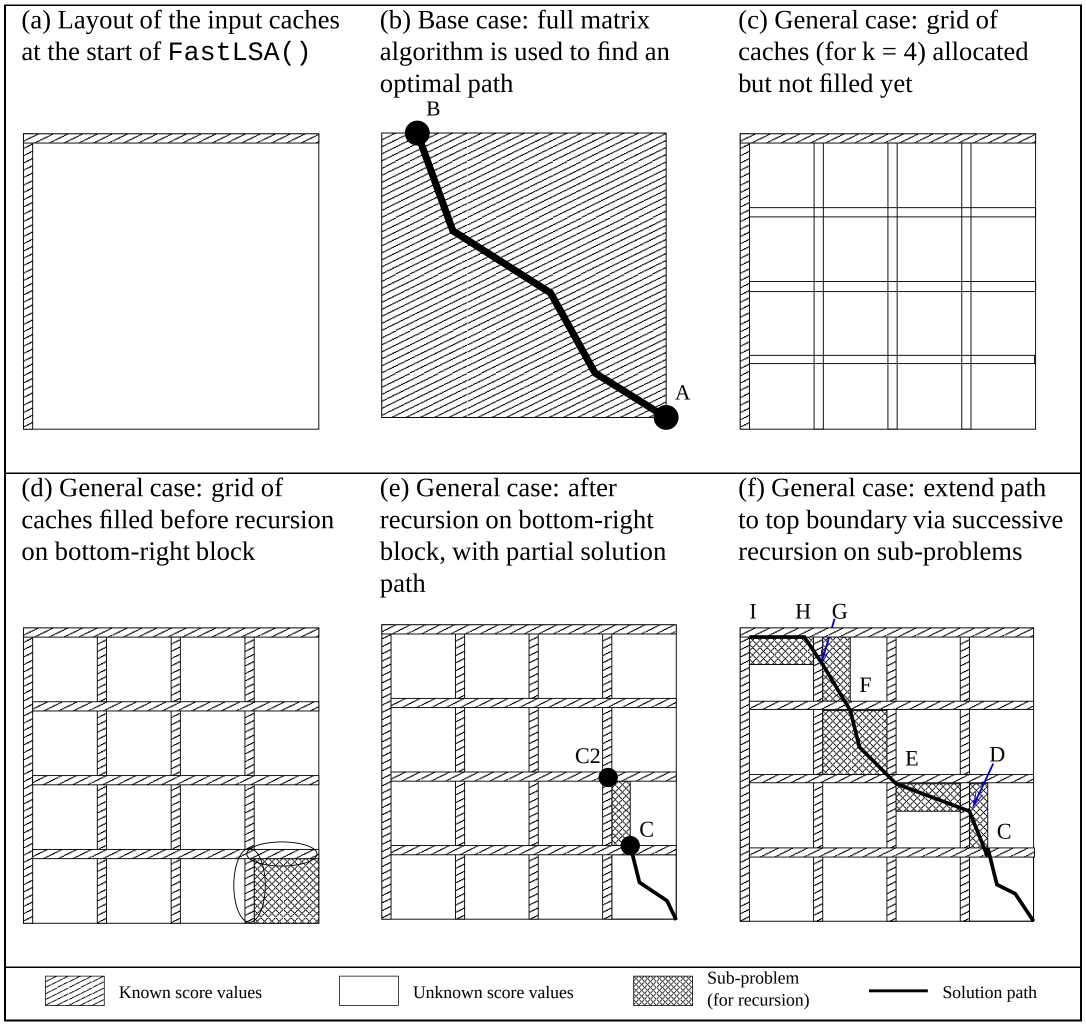

(a) Layout of the input caches at the start of FastLSA()
(b) Base case: full matrix algorithm is used to find an optimal path
B
A
(c) General case: grid of caches (for k = 4) allocated but not filled yet
(d) General case: grid of caches filled before recursion on bottom-right block
(e) General case: 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 3: Execution Stages of FastLSA

The first goal of the general case is to find the values of the entries of the DPM 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 DPM. The grid *flsaGrid* is allocated in order to store these values once they are computed (line 3 in Figure 2). *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 DPM.

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). Figure 3(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 DPM are computed, except for those forming the bottom-right sub-matrix. This is accomplished with the call to *fillGridCache* (line 5 in Figure 2). 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 3(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 in Figure 2). 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 3(e)). Note that *C* could also have been on the left boundary of the bottom-right sub-matrix. Naturally, it does not matter whether the bottom-right sub-matrix is a base case or requires its own recursive calls.

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 DPM. This step is accomplished through successive recursive calls to *FastLSA* in the `while`-loop (lines 9–13 in Figure 2).

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 subproblem to be solved by *FastLSA* is identified through a call to *UpLeft* (line 10 in Figure 2). The coordinates of this new logical DPM 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 3(e) shows the logical DPM 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 DPM to North and West are identified. These are the new caches which are passed with the recursive call to *FastLSA* as *newCacheRow* and *newCacheColumn*. When this recursive call returns, the optimal path for the original problem has been extended from *C* to the entry *D* (Figure 3(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 3(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 DPM 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 DPM. Figure 3(f) shows the partial optimal path being extended to the top-left corner *I* along the first row of the DPM. The resulting optimal path corresponds uniquely to an optimal alignment between the input sequences a and b.

It is useful to observe that FastLSA solves a succession of rectangular problems, called *FastLSA sub-problems*, 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 DPM, 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 of Charter, Schaeffer, and Szafron [4]. 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 DPM 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 initial call uses the General Case of the algorithm, then $k-1$ 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, in the worst case, we get:

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

The worst-case recursive relation for space becomes

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

Because the space for Base Case subproblems is allocated in the Base Case buffer, it is true that $S(\frac{m}{k^a},\frac{n}{k^a},k) \le BM$, and Equation 2 becomes

$$\begin{aligned}
S(m,n,k) &\le (k-1)\times(m+n)\times\frac{1-\frac{1}{k^a}}{1-\frac{1}{k}} + BM \\
&= k\times(m+n)\times(1-\tfrac{1}{k^a}) + BM \\
&\le k\times(m+n) + BM.
\end{aligned} \tag{3}$$

Equation 3 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 DPM 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 that, in the worst case scenario,

$$T(m,n,k) = m\times n\times\tfrac{k+1}{k-1}. \tag{4}$$

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 $k=5$, $T(m,n,5) = 1.5\times m\times n$. The upper bound provided by FastLSA decreases when the value of k increases.

Now that we have established reasonable upper bounds on the space (Equation 3) and time (Equation 4) complexity for FastLSA, we can now consider the performance of the algorithm in practice.

4 Experimental Results for Sequential FastLSA

We compared the empirical performance of the FM algorithm, Hirschberg's algorithm, and FastLSA using a common software and hardware base. The commercial ChromaTool sequence analysis suite developed by BioTools, Inc. (`www.biotools.com`) uses an implementation of FastLSA. For completeness, we implemented an FM algorithm and Hirschberg's algorithm within the same BioTools framework. All algorithms have been tuned for performance including the removal of a number of error checking code segments. Also, all algorithms share the same input/output code, the same scoring table (Table 1 is a sub-table), and a gap penalty where creating a new gap has a value of -20 and extending an existing gap has a value of -10 (i.e., an affine gap penalty). The experiments were performed on a 800 MHz Pentium III (Coppermine) with 16 Kbytes of Level 1 data cache, 256 Kbytes of Level 2 cache (clocked at 800 MHz), 133 MHz front side bus (FSB), 512 MB of main memory and Red Hat Linux 6.1 with the Linux 2.2.16 kernel. Although there are two CPUs, our application is single-threaded.

We mimic a typical sequence search that takes a new query amino acid or DNA sequence and pairwise aligns it with each sequence in a database. High alignment scores between the query sequence and a specific database sequence are flagged for further consideration by the biologist. Given that these pairwise alignments produce optimal matches for the selected scoring function, the speed of these pairwise alignments is the most important consideration.

We randomly selected 5 sequences of lengths 100, 200, 500, 800, 1000, and 2000 amino acids, plus or minus 5% in length, from the Swiss-Prot database [2] to serve as our query sequences. The results of our first experiment are shown in Table 2. Note that, with one exception, FastLSA is the fastest algorithm.

Since 5 sequences with the same nominal length are used as the query sequences for the experiment,

Query Length	Full Matrix	Hirschberg	FastLSA
100	0.307 $\pm$ 0.003	0.389 $\pm$ 0.007	**0.262 $\pm$ 0.004**
200	0.621 $\pm$ 0.008	0.885 $\pm$ 0.014	**0.595 $\pm$ 0.009**
500	**1.594 $\pm$ 0.016**	2.551 $\pm$ 0.042	1.713 $\pm$ 0.028
800	2.594 $\pm$ 0.049	3.853 $\pm$ 0.129	**2.580 $\pm$ 0.081**
1000	3.216 $\pm$ 0.026	4.305 $\pm$ 0.048	**2.882 $\pm$ 0.030**
2000	6.531 $\pm$ 0.091	9.418 $\pm$ 0.642	**6.136 $\pm$ 0.415**

Table 2: Sequential Search of the Swiss-Prot Databases with FM, Hirschberg and FastLSA (times in $seconds \times 10^3$, fastest times are in boldface)

---

there are a total of 30 query sequences from 6 categories based on length. The average time for the 5 query sequences of similar length is given in the figure and the error is one standard deviation of the 5 data points. All of the algorithms used the same query sequences and the same version of the Swiss-Prot database. We used Equation 5 to pick a value of k, based on the lengths of the two sequences. This formula has been empirically determined to obtain good results.

$$k = truncate(log10(m)) + truncate(log10(n)) + 3 \tag{5}$$

We used a heuristic function from the ChromaTool code for picking the buffer size of the recursion-terminating call to our FM code.

Based on the complexity analysis, one would expect FM to be the fastest algorithm in all cases. After all, FM does not require any re-computation to recover the path of the optimal alignment. In contrast, both Hirschberg's algorithm and the FastLSA reduce their storage costs at the expense of re-computation. In fact, FM does 0 re-computations, Hirschberg's algorithm does $m \times n$ re-computations and FastLSA does $(m \times n)/8$ re-computations (for $k = 8$). For example, if the query sequence has size 100 and the database sequences range in size from 100 to 5,000, FM does 0 re-computations, FastLSA does 1250 to 62,500 re-computations, and Hirschberg's algorithms does 10,000 to 500,000 re-computations for each alignment.

Since FastLSA makes fewer re-computations than Hirschberg's algorithm, it is not surprising that it is consistently faster. However, why is FastLSA faster than FM for query sequences of length 100 and 200, slower than FM for sequences of size 500 and then faster again for longer sequences?

An inescapable fact of contemporary computer systems is that, in practice, the cache behavior of an algorithm can have a substantial impact on its performance. Each query sequence of size 100 was aligned against the entire Swiss-Prot database, which contains sequences ranging from less than 100 amino acids to over 5,000 amino acids. This means that the DPM ranged in size from $100 \times 100 \times 4$ bytes = 40 Kbytes to $100 \times 5000 \times 4$ bytes = 2 Mbytes. Since the secondary cache has only 256 Kbytes, the FM DPM would not fit in secondary cache and a large number of main memory accesses were made. In contrast, the memory requirements for FastLSA are much smaller. FastLSA with $k = 8$ requires only $8 \times (100 + 1000) \times 16$ bytes = 140.8 Kbytes for the grid vectors. This easily fits into the 256 Kbyte secondary cache. Since a main memory access is more than 10 times slower than an access to secondary cache, the FM DPM not fitting into cache is sufficient to account for the faster FastLSA performance. Hirschberg's algorithm also fits into the secondary cache. However, since it does more re-computations than FastLSA, it cannot overtake the FM algorithm.

However, Table 2 does not present the whole story. There is a sequence length for which FM will exhaust main memory and page to disk. This is a disastrous situation for the algorithm since disk access time is more than a million times greater than memory access time. At this point, FastLSA and Hirschberg will again dominate FM and by a significantly larger margin. For the main memory configuration used in our experiments (512Mbytes) this does not occur using the Swiss-Prot database, even with a query sequence of size 5,000 since the longest database target sequences have length 5,000.

From Table 2 we conclude that for shorter sequences, the choice of the best algorithm depends on various cache effects. However, FastLSA is always better than Hirschberg's algorithm. In the typical case of comparing query sequences against a database, such as Swiss-Prot, with sequences of various lengths, FastLSA is usually faster than FM due to good caching for short sequences and no paging for longer sequences. However, for a very narrow range of intermediate length sequences when all three algorithms are out of cache, but none of the algorithms exhibit paging, FM and FastLSA have very similar performance. This is illustrated by the 500 and 800 query lengths in Table 2.

5 Parallel FastLSA

Sequential FastLSA outperforms other sequential pairwise alignment algorithms (Table 2). This makes FastLSA a reasonable candidate for parallelization (i.e., parallelize the best available sequential algorithm).

Given the large DNA sequences (e.g., tens of thousands of bases) that some researchers wish to study [6, 19, 7], a parallel FastLSA may be highly desirable. In particular, 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.

5.1 Description of the Parallel FastLSA Algorithm

Before we provide both theoretical and empirical analysis of Parallel FastLSA, we describe the parallel algorithm and discuss some implementation issues.

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), 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).

The pseudo-code for Parallel FastLSA is shown in Figure 4. 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 (Section 6.3). 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 section we analyze the performance of an implementation 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 parallel implementation more 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 previously explained, FastLSA stops recursing when the input logical DPM *flsaProblem* can be allocated in the Base Case buffer (line 1 in Figure 4). 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 DPM 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 DPM. In order to compute the value of a DPM entry, the values of the adjacent entries from North, West, and North–West must be available.

The DPM is logically partitioned in $R \times C$ equally sized rectangular regions, with $R \geq 1$ and $C \geq 1$. Note that in Figure 5, $R = 8$ 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

```
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 3.6 (a) */
1   if flsaProblem fits in allocated buffer then
        // BASE CASE
        /* Figure 3.6 (b) */
2       return parallelSolveFullMatrix( flsaProblem, cacheRow, cacheColumn, flsaPath )

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

    /* Figure 3.6 (c) */
5   parallelFillGridCache( flsaProblem, flsaGrid )

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

    /* Figure 3.6 (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 3.6 (e) */
13      flsaPathExt = ParallelFastLSA( flsaSubProblem, newCacheRow, newCacheColumn, flsaPathExt )

14  deallocateGrid( flsaGrid )

    /* Figure 3.6 (f) */
15  return flsaPathExt
```

Figure 4: Pseudo-Code for Parallel FastLSA

the DPM, 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 5. 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.

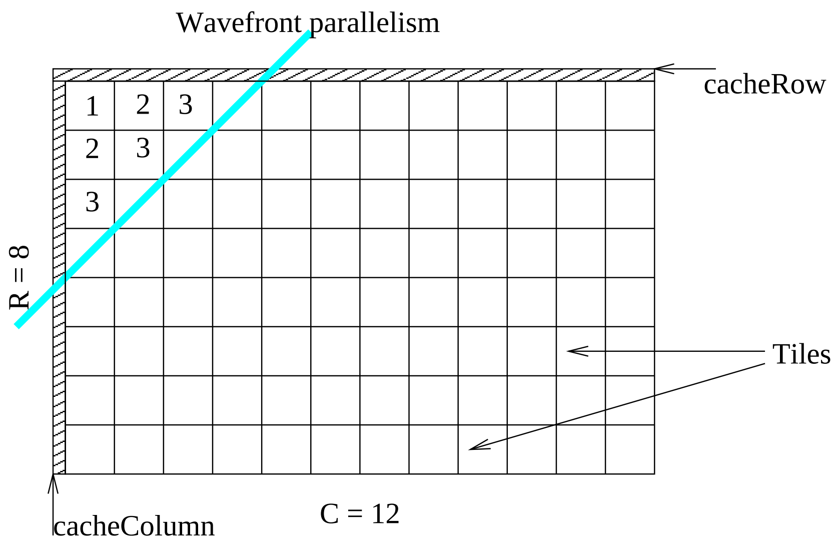


Figure 5: Data Partitioning for Parallel Base Case Subproblems

After the first step, there is enough information available to start computing the entries in the tiles which neighbor the top-left tile to the East and the 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 5, 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 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 DPM 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 DPM 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 earlier. However, the Fill Cache subproblems are much larger

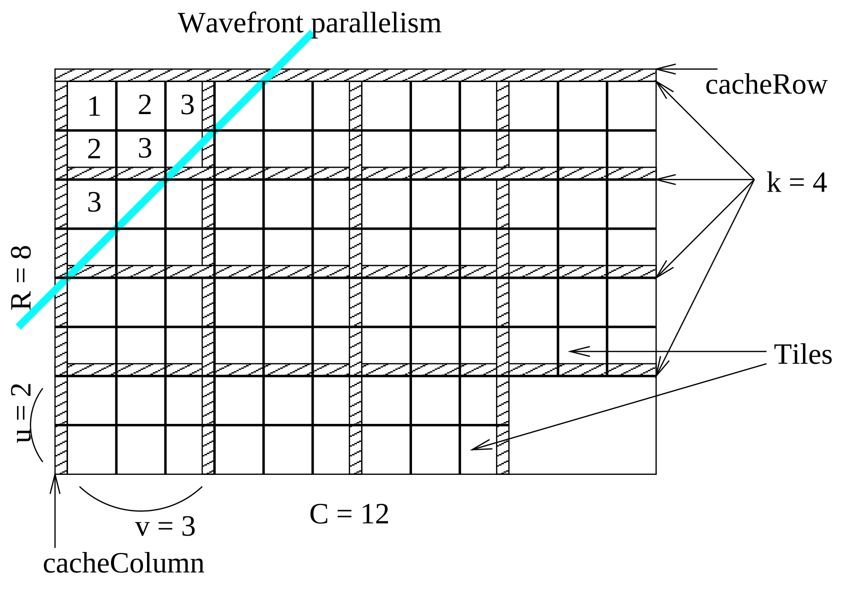


Figure 6: Data Partitioning for Parallel Fill Cache Subproblems

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 6). The result is a grid of finer granularity than the FastLSA grid. This new grid partitions the DPM 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 6, $k = 4$, $u = 2$, 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. 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 7). 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 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

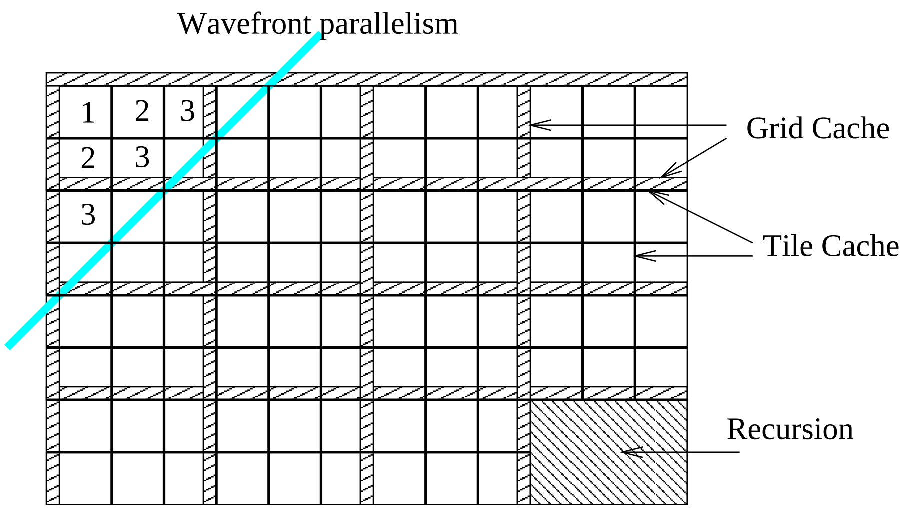


Figure 7: FastLSA Grid Cache and Tile Cache for Parallel Fill Cache Subproblems

neighbor the top-left tile (i.e., the tiles labeled with 2 in Figure 7). 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 7 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 7, 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 7). Note that new caches of each type, FastLSA Grid Cache and Tile Cache, are allocated in shared memory for each Fill Cache subproblem solved.

5.2 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.

5.3 Dynamic Distribution of Work

Initially, only the top-left tile, which is labelled 1 in Figure 7, 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 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 7 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

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

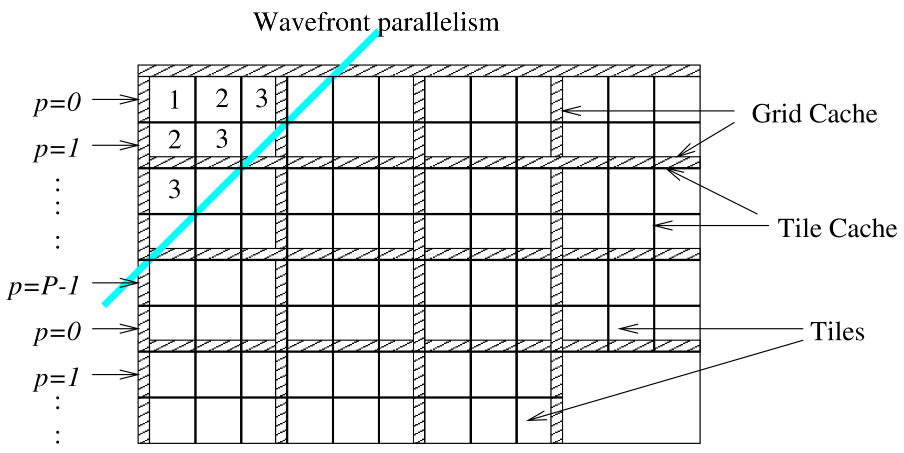


Figure 8: Parallel FastLSA: Static Distribution of Work

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 larger than the dimensions of the input DPM.

5.4 Static Distribution of Work

Static work distribution is another solution to the problem of allocating the tiles, which is dependent in a wavefront manner, to the P processors available. As shown in Figure 8, each of the R rows of tiles is assigned to a processor in a circular or round-robin 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 8 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 of Martins *et al.* [13]. 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 coherent caches that support inexpensive spinning for reads. Each time the processor p finishes solving a tile, an index is incremented, and the processor $p + 1$ $(\bmod\ P)$ 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.

5.5 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 subsection focuses on the derivation of the space and time expressions that are upper bounds for the space and time complexity of Parallel FastLSA.

5.5.1 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:

$$\begin{aligned} PFastLSA(m,n,k) = PFillCache(m,n,k), PFastLSA(\tfrac{m}{k}, \tfrac{n}{k}, k), \\ PFastLSA(m_1, n_1, k), \ldots, PFastLSA(m_z, n_z, k); \end{aligned} \tag{6}$$

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 4). 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 have been provided by Charter, Schaeffer, and Szafron [4].

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

$$\frac{m}{k^a} \times \frac{n}{k^a} \leq BM < \frac{m}{k^{a-1}} \times \frac{n}{k^{a-1}}. \tag{7}$$

This is equivalent to

$$a-1 < \frac{\log \frac{m \times n}{BM}}{2 \log k} \leq a \Leftrightarrow \left\lceil \frac{\log \frac{m \times n}{BM}}{2 \log k} \right\rceil = a. \tag{8}$$

5.5.2 Space Complexity

Definition 1 *Let $S(m,n,k)$ be the maximum number of DPM 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*

$$S(m,n,k) \leq (3k-1) \times (m+n) + \frac{P}{C} \times n + R \times C - u \times v + BM. \tag{9}$$

Please see Appendix A for the proof.

5.5.3 Time Complexity

Definition 3 *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 4 *Let $WT(m, n, k, P)$ be defined as in Definition 3. 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*

$$WT(m, n, k, P) \leq \tfrac{m \times n}{P} \times (1 + \tfrac{P^2 - P}{R \times C}) \times (\tfrac{k}{k-1})^2. \tag{10}$$

Please see Appendix A for the proof.

The previous discussion 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.

6 Experimental Results for Parallel FastLSA

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 [4]. For simplicity, the FastLSA implementations that we benchmark find the globally optimal alignment of two sequences using a straightforward scoring function where all identical matches have a score of 2, all mismatches have a score of -1, and the gap penalty is -2.

We discuss 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 [16]. Most of their examples are comparisons of "some region of the human genome with the synthetic region from a rodent genome" [18]. 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 [18]. 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*) [18] 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" [17]. The human sequence is 319,030 bp long, and the mouse sequence is 305,636 bp long.

Throughout the benchmarking process discussed in this section, all parameters introduced in Section 5.1

	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 DPM	$37,349 \times 37,785$	*XRCC1*;
		$55,820 \times 66,315$	*Myosin*;
		$305,636 \times 319,030$	*TCR*.

Table 3: The Parameters which Influence the FastLSA algorithms

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 3 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, independent 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 DPM is divided in $3 \times k$ rows and $4 \times k$ 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 section are obtained using an implementation based on the Dynamic Distribution of Work strategy. This strategy of work distribution is introduced in Section 5.3. 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 section solves the Base Case subproblems sequentially. This modified version of Parallel FastLSA is preferred to the one described in Section 5.1 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 6.3 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 [21]). 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 DPM 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 section 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 higher than in reality.

6.1 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 a best value for k, given the size of the sequences and P, the number of processors used. This best value would ensure that the overall alignment time is close to the theoretical optimal time. However, the relationship between the best value of k, P, and the size of the

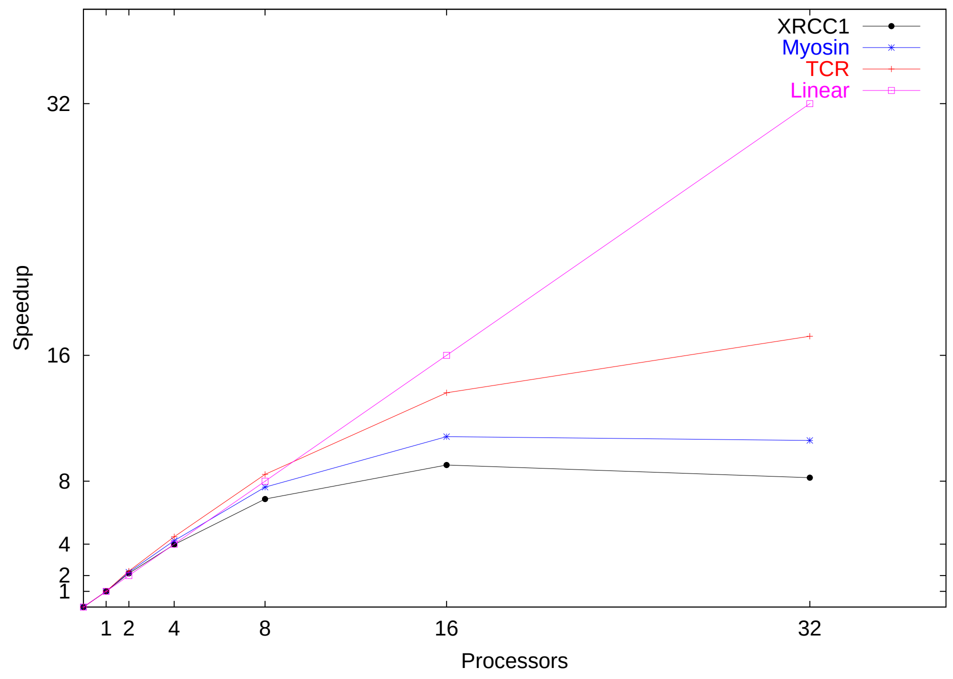


Figure 9: Best Speedups for XRCC1, Myosin, and TCR

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 best 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 the best values. This interval includes an empirical best value for k in most of the combinations benchmarked.

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 9. Table 4 shows the execution time for each sequence alignment performed and the corresponding value for k that achieved that performance. Note that the largest problem (i.e., *TCR*) requires over 5,040 seconds (i.e., 1.4 hours to align), which suggests the need for efficient parallel algorithms to tackle even larger sequences [6, 19, 7].

For the pair of short sequences, *XRCC1*, the speedup is linear for 2 and 4 processors, but starts deteri-

Sequences	**Number of Processors**	**Time (sec.)**	**Speedup**	**Best 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: Real Times, Speedups, and k

orating 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 DPM computed for the *Myosin* sequences has 2.6 times more entries than the DPM computed for the *XRCC1* sequences. The larger *Myosin* DPM 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 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 [11].

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.

6.2 Case Study: Myosin Dataset

In order to understand how the parameters and the design of Parallel FastLSA influence its execution time, we perform a detailed empirical analysis of the performance of the algorithm. We select the Myosin dataset, which is the moderate-sized dataset, for this case study.

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. Since there are thousands of these subproblems for each sequence pair, the statistical distribution of the subproblem execution times are presented. 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

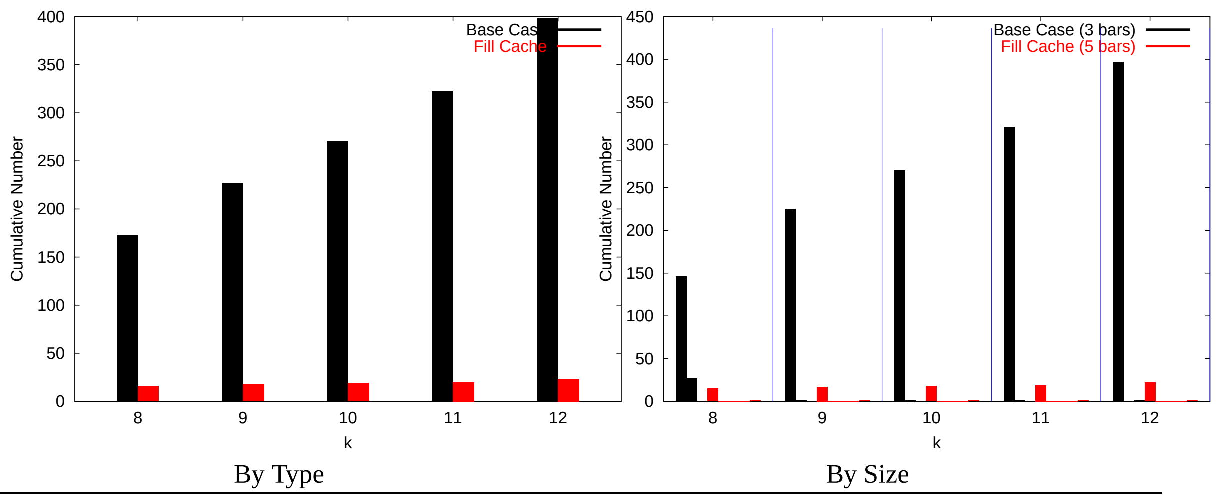


Figure 10: FastLSA Subproblem Count: Parallel FastLSA Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Type/Size of the FastLSA Subproblems)

graphs obtained are presented in the following three sections.

6.2.1 Subproblem Count Graph

A subproblem count graph (Figure 10) 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. This graph (Figure 10) 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.

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 DPM 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 DPM 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. Depending on the specific input data and other system parameters, some of the subsets (and, thus, bars) may be empty.

The plot for the size-based partitioning shows (up to) eight bars for each value of k. The (up to) three black bars on the left indicate the number of FastLSA subproblems in the Base Case partitions, while the (up to) 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.

From this empirical analysis (Figure 10), we can see that Base Case subproblems dominate the run-time behaviour of the algorithm in terms of the *number* of problem instances. But, in the next section, we examine how the Fill Cache subproblems actually dominate in terms of the *time* spent in the computation. Of course, as per Amdahl's Law, the benefits of parallelism come from parallelizing the Fill Cache subproblems. In fact, Section 6.3 argues that solving the Base Case subproblems sequentially (instead of in parallel) results in better overall speedups because the Base Case problems are too small in their granularity of work.

6.2.2 Execution Time Graph

Trends in execution time are among the most important indicators of the performance of an algorithm. Figure 11 presents the execution times for the Parallel FastLSA algorithm. A series of six graphs shows the changes in execution time as the number of processors varies. When the FastLSA subproblems are clustered based on their type (Figure 11), 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. 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.

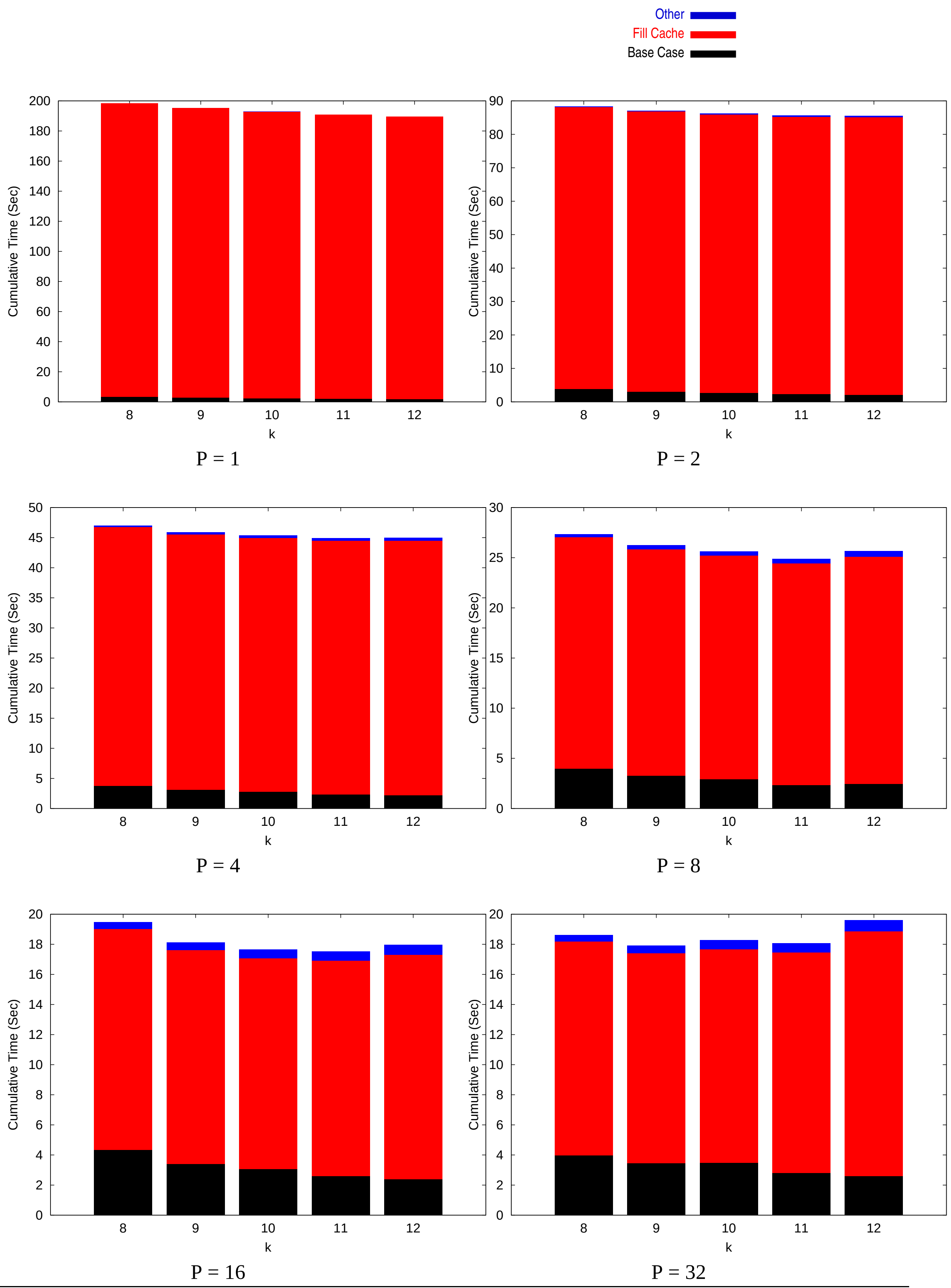
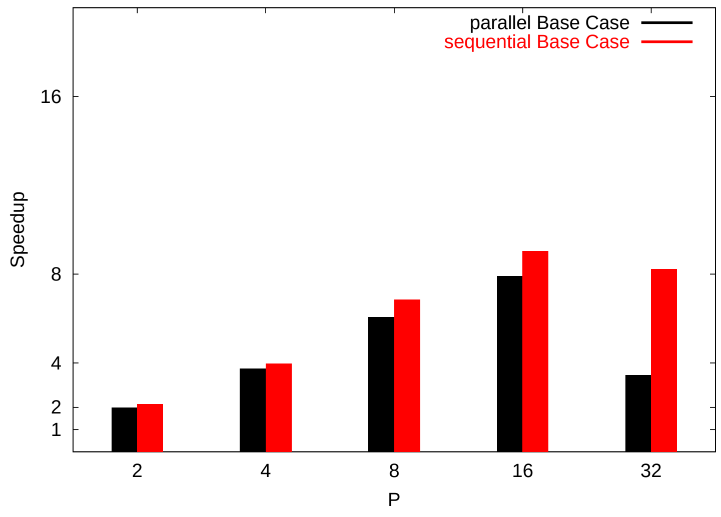
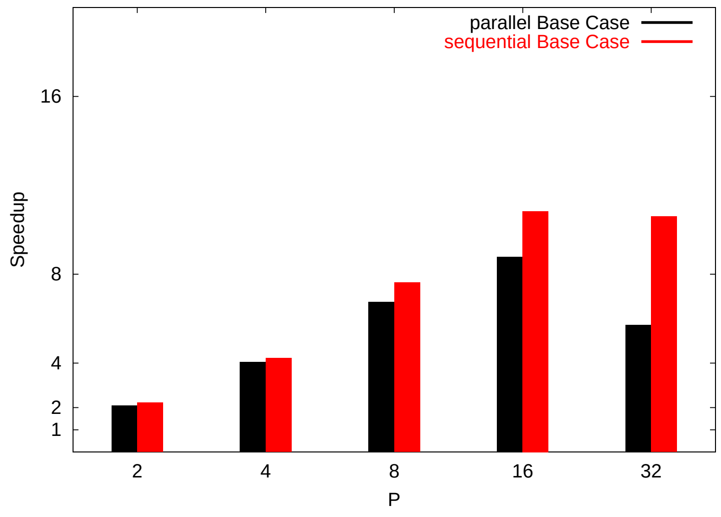


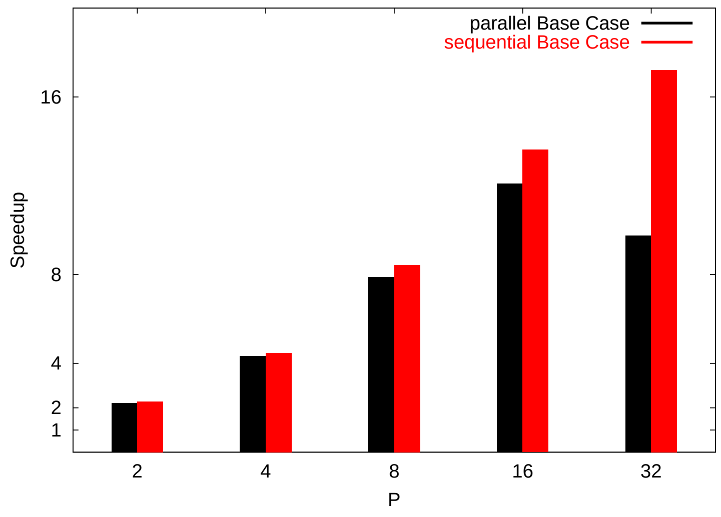
Figure 11: Execution Time: Parallel FastLSA Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Type of the FastLSA Subproblems)



XRCC1



Myosin



TCR

Figure 12: Comparison of the overall speedups for the two versions of Parallel FastLSA

Figure 11 exposes a trend in the values of the total execution time across the five values of k: the best 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 DPM 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.

6.3 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 12, 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.

7 Concluding Remarks

Sequence alignment is a fundamental operation for homology search in bioinformatics. For two DNA or protein sequences of length m and n, full-matrix (FM), dynamic programming alignment algorithms such as Needleman-Wunsch and Smith-Waterman take O($m \times n$) time and use a possibly prohibitive O($m \times n$) space. Hirschberg's algorithm reduces the space requirements to O($min(m, n)$), but requires approximately twice the number of operations required by the FM algorithms.

The Fast Linear Space Alignment (FastLSA) algorithm adapts to the amount of space available by trading space for operations. What makes FastLSA unique is its parameter k, which can be used to tune its storage requirements for a given amount of cache memory or main memory. Our experiments show that, in practice, due to memory caching effects, FastLSA is preferred over the Hirschberg and the FM algorithms. To further improve the performance of FastLSA, we have parallelized it using a simple but effective form of wavefront parallelism. 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.

Again, a recurring theme in this paper is the importance of algorithms that can be parameterized and tuned to take advantage of cache memory and main memory sizes. Existing algorithms for sequence alignment (i.e., FM and Hirschberg) cannot be similarly parameterized. Furthermore, the selected value for parameter k has a significant impact on the parallel speedups of the algorithm, which results in interesting lessons in performance trade-offs. For example, large values of k reduce the amount of re-computation and increase the performance of FastLSA. Larger values of k may also help FastLSA better exploit processor caches for greater performance. However, large values of k can also reduce the granularity of work in Parallel FastLSA, which is detrimental to performance. For future work, more empirical results are required to develop better guidelines for selecting a value of k that best exploits caches, main memory sizes, **and** provides good parallel speedups. Another future direction is to modify Parallel FastLSA such that the granularity of work is independent of k.

Given the large DNA sequences (e.g., tens of thousands of bases) that some researchers wish to study [6, 19, 7], the space and time complexity of a sequence alignment algorithm become increasingly important. The combination of FastLSA's parameterized storage complexity, good analytical time complexity, easy parallelization, and excellent empirical performance makes FastLSA a good choice for pairwise sequence alignment.

8 Acknowledgments

We would like to acknowledge Scott Fortin at BioTools for several helpful discussions and making their source code available to us. This research was partially funded by research grants from the Protein Engineering Network of Centres of Excellence (PENCE), the Natural Sciences and Engineering Research Council of Canada (NSERC), the Alberta Informatics Circle of Research Excellence (iCORE) and the Canada Foundation for Innovation (CFI).

References

[1] C.E.R. Alves, E.N. Cáceres, F. Dehne, and S.W. Song. Parallel Dynamic Programming for Solving the String Editing Problem on a CGM/BSP. In *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 275–281, Winnipeg, Manitoba, Canada, August 10–13, 2003.

[2] R. D. Appel, A. Bairoch, and D. F. Hochstrasser. A new generation of information retrieval tools for biologists: the example of the ExPASy WWW server. *Trends in Biochem. Sci.*, 19:258–260, 1994.

http://ca.expasy.org/sprot/.

[3] P.G. Bradford. *Parallel Dyanmic Programming*. PhD thesis, Department of Computer Science, Indiana University, 1994.

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

[5] M. O. Dayhoff, W. C. Barker, and L. T. Hunt. Establishing homologies in protein sequences. *Methods in Enzymology*, 91:524–545, 1983.

[6] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[7] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.

[8] A. Driga. Parallel FastLSA: A parallel algorithm for pairwise sequence alignment. Master's thesis, University of Alberta, 2002.

[9] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons. FastLSA: A Fast, Linear-Space, Parallel and Sequential Algorithm for Sequence Alignment. In *Proc. 32nd International Conference on Parallel Processing (ICPP)*, pages 48–57, Kaohsiung, Taiwan, October 6–9, 2003. Available at http://www.cs.ualberta.ca/˜paullu/.

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

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

[12] G. Lewandowski, A. Condon, and E. Bach. Asynchronous Analysis of Parallel Dynamic Programming Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):425–438, 1996.

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

[14] E. Myers and W. Miller. Optimal alignments in linear space. *CABIOS*, 4:11–17, 1988.

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

[16] Penn State University. Bioinformatics Group. http://bio.cse.psu.edu, 2001.

[17] Bioinformatics Group Penn State University. TCR sequences. http://bio.cse.psu.edu/pipmaker/examples.html, 2001.

[18] Bioinformatics Group Penn State University. XRCC1 and Myosin sequences. http://globin.cse.psu.edu/globin/html/pip/examples.html, 2001.

[19] N. T. Perna and et al. Genome sequence of enterohaemorrhagic *Escherichia coli O157:H7*. *Nature*, 409(6819):529–533, 2001.

[20] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[21] Veridian Systems. PBS. http://www.pbspro.com, 2001.

Contents

1 Introduction **3**

1.1 Background 3

2 Related Work **7**

2.1 Dynamic Programming and Full-Matrix Algorithms 7

2.2 Hirschberg's Algorithm 9

2.3 Parallel Dynamic Programming 10

3 Sequential FastLSA Algorithm **11**

4 Experimental Results for Sequential FastLSA **19**

5 Parallel FastLSA **22**

5.1 Description of the Parallel FastLSA Algorithm 22

5.2 Implementation Details 28

5.3 Dynamic Distribution of Work 28

5.4 Static Distribution of Work 30

5.5 Space and Time Complexity 31

5.5.1 FastLSA Recursion Pattern 31

5.5.2 Space Complexity 32

5.5.3 Time Complexity 33

6 Experimental Results for Parallel FastLSA **33**

6.1 General Observations 37

6.2 Case Study: Myosin Dataset 40

6.2.1 Subproblem Count Graph 41

6.2.2 Execution Time Graph 42

6.3 Base Case Subproblems: Sequential Approach versus Parallel Approach 45

7 Concluding Remarks **46**

8 Acknowledgments **47**

A Proofs **51**

A Proofs

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

Theorem 2 (from Section 5.5.2) *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*

$$S(m,n,k) \leq (3k-1) \times (m+n) + \tfrac{P}{C} \times n + R \times C - u \times v + BM. \tag{11}$$

Proof. For an algorithm trace such as that in Equation 6,

$$\begin{aligned} S(m,n,k) &= \text{maxSpace}(PFastLSA(m,n,k)) \\ &= \max\Big(\text{maxSpace}(PFillCache(m,n,k)), \\ &\quad GridSpace(m,n,k) + \text{maxSpace}(PFastLSA(\tfrac{m}{k},\tfrac{n}{k},k)), \\ &\quad GridSpace(m,n,k) + \text{maxSpace}(PFastLSA(m_1,n_1,k)), \ldots, \\ &\quad GridSpace(m,n,k) + \text{maxSpace}(PFastLSA(m_z,n_z,k))\Big). \end{aligned} \tag{12}$$

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:

$$\text{maxSpace}(PFastLSA(\tfrac{m}{k},\tfrac{n}{k},k)) \geq \text{maxSpace}(PFastLSA(m_i,n_i,k)), \forall i, 1 \leq i \leq z. \tag{13}$$

Equation 12 becomes

$$\begin{aligned} S(m,n,k) = \max\Bigg(& \text{maxSpace}(PFillCache(m,n,k)), \\ & GridSpace(m,n,k) + \text{maxSpace}(PFastLSA(\tfrac{m}{k}, \tfrac{n}{k}, k))\Bigg) \\ = \max\Bigg(& \text{maxSpace}(PFillCache(m,n,k)), GridSpace(m,n,k) + S(\tfrac{m}{k}, \tfrac{n}{k}, k)\Bigg). \end{aligned} \tag{14}$$

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. In summary,

$$\begin{aligned} \text{maxSpace}(PFillCache(m,n,k)) &= A(m,n,k) = (k-1)(m+n) + (k-1)(m+n)+ \\ &+ (m+n) + R \times C - u \times v + P\tfrac{n}{C} \\ &= (2k-1)(m+n) + R \times C - u \times v + \tfrac{P}{C}n, \end{aligned} \tag{15}$$

and

$$GridSpace(m,n,k) = (k-1)(m+n). \tag{16}$$

Using the previous two equations, Equation 14 becomes

$$\begin{aligned}
S(m,n,k) &= \max\Big(A(m,n,k),(k-1)(m+n)+S(\tfrac{m}{k},\tfrac{n}{k},k)\Big)\\
&= \max\Big(A(m,n,k),(k-1)(m+n)+\\
&+\max\Big(A(\tfrac{m}{k},\tfrac{n}{k},k),(k-1)\tfrac{m+n}{k}+S(\tfrac{m}{k^2},\tfrac{n}{k^2},k)\Big)\Big)\\
&= \max\Big(A(m,n,k),\max\Big((k-1)(m+n)+A(\tfrac{m}{k},\tfrac{n}{k},k),\\
&(k-1)(m+n)+(k-1)\tfrac{m+n}{k}+S(\tfrac{m}{k^2},\tfrac{n}{k^2},k)\Big)\Big)\\
&= \max\Big(\max\Big(A(m,n,k),(k-1)(m+n)+A(\tfrac{m}{k},\tfrac{n}{k},k)\Big),\\
&(k-1)(m+n)(1+\tfrac{1}{k})+S(\tfrac{m}{k^2},\tfrac{n}{k^2},k)\Big).
\end{aligned} \tag{17}$$

In order to unwind the recursive formula from Equation 17, the result of Lemma 5 is used. Note that the statement and proof of Lemma 5 immediately follows this proof. Lemma 5 states that if $A(m,n,k)$ is defined as in Equation 15, then

$$A(m,n,k) \geq (k-1)(m+n)(1+\tfrac{1}{k}+\cdots+\tfrac{1}{k^{j-2}})+A(\tfrac{m}{k^{j-1}},\tfrac{n}{k^{j-1}},k), \forall j, 2 \leq j \leq a. \tag{18}$$

For example, for $j=2$, the inequality of Lemma 5,

$$A(m,n,k) \geq (k-1)(m+n)+A(\tfrac{m}{k},\tfrac{n}{k},k), \tag{19}$$

can be rewritten as

$$\max\Big(A(m,n,k),(k-1)(m+n)+A(\tfrac{m}{k},\tfrac{n}{k},k)\Big)=A(m,n,k). \tag{20}$$

By rewriting the inequalities of Lemma 5 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:

$$\begin{aligned} S(m,n,k) &= \max\left(A(m,n,k), (k-1)(m+n)(1+\tfrac{1}{k}) + S(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k)\right) \\ &= \cdots = \\ &= \max\left(A(m,n,k), (k-1)(m+n)(1+\tfrac{1}{k}+\cdots+\tfrac{1}{k^{a-1}}) + S(\tfrac{m}{k^a}, \tfrac{n}{k^a}, k)\right) \\ &= \max\left(A(m,n,k), k(m+n)(1-\tfrac{1}{k^a}) + S(\tfrac{m}{k^a}, \tfrac{n}{k^a}, k)\right). \end{aligned} \tag{21}$$

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

$$\begin{aligned} \max\left(A(m,n,k), k(m+n)(1-\tfrac{1}{k^a}) + BM\right) &\leq A(m,n,k) + k(m+n)(1-\tfrac{1}{k^a}) + BM \\ &\leq A(m,n,k) + k(m+n) + BM = \\ &= (2k-1)(m+n) + R \times C - u \times v + \\ &+ \tfrac{P}{C}n + k(m+n) + BM \\ &= (3k-1) \times (m+n) + \tfrac{P}{C}n + \\ &+ R \times C - u \times v + BM. \end{aligned} \tag{22}$$

Therefore,

$$S(m,n,k) \leq (3k-1) \times (m+n) + \tfrac{P}{C} \times n + R \times C - u \times v + BM, \tag{23}$$

which concludes the proof of Theorem 2. ■

Lemma 5 *Let $A(m,n,k)$ be defined as in Equation 15. Then*

$$A(m,n,k) \geq (k-1)(m+n)(1+\tfrac{1}{k}+\cdots+\tfrac{1}{k^{j-2}}) + A(\tfrac{m}{k^{j-1}}, \tfrac{n}{k^{j-1}}, k), \forall j, 2 \leq j \leq a. \tag{24}$$

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

$$\begin{aligned} A(m,n,k) \geq (k-1)(m+n)(1+\tfrac{1}{k}+\cdots+\tfrac{1}{k^{j-2}}) + A(\tfrac{m}{k^{j-1}}, \tfrac{n}{k^{j-1}}, k) &\Leftrightarrow \\ A(m,n,k) - A(\tfrac{m}{k^{j-1}}, \tfrac{n}{k^{j-1}}, k) \geq k(m+n)(1-\tfrac{1}{k^{j-1}}) &\Leftrightarrow \\ (2k-1)(m+n)(1-\tfrac{1}{k^{j-1}}) + \tfrac{P}{C}n(1-\tfrac{1}{k^{j-1}}) \geq k(m+n)(1-\tfrac{1}{k^{j-1}}). & \end{aligned} \tag{25}$$

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

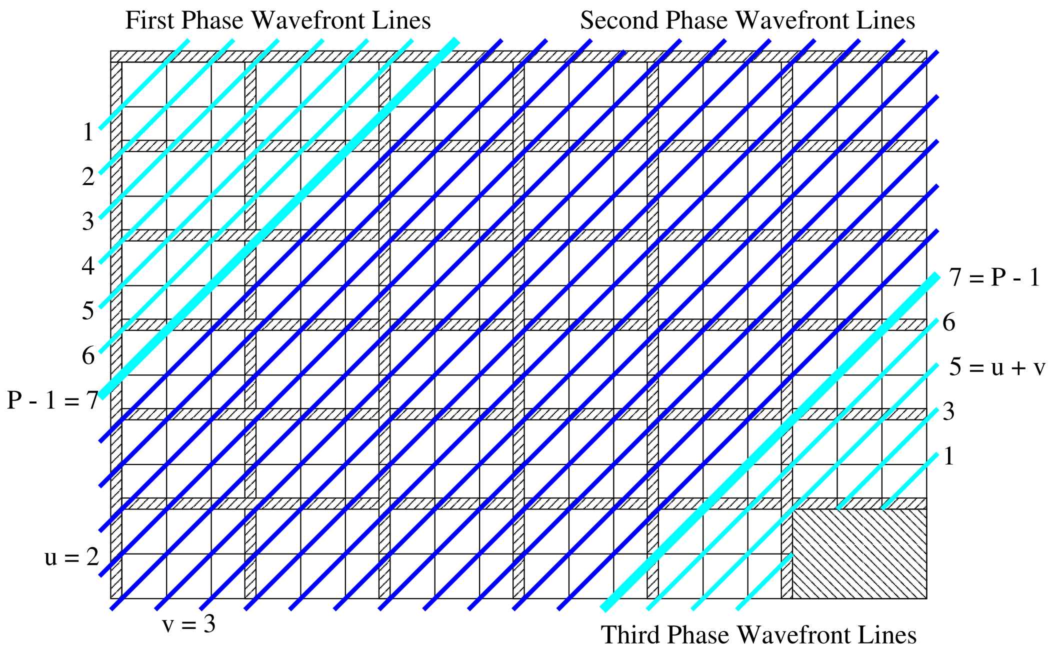
$$\begin{aligned} (2k-1)(m+n)(1-\tfrac{1}{k^{j-1}}) \geq k(m+n)(1-\tfrac{1}{k^{j-1}}) &\Leftrightarrow \\ (2k-1) \geq k &\Leftrightarrow \\ k \geq 1, & \end{aligned} \tag{26}$$

which is true. Therefore, the inequality of Lemma 5 is true $\forall j, 2 \leq j \leq a$. ■

Theorem 4 (from Section 5.5.3) Let $WT(m,n,k,P)$ be defined as in Definition 3. 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

$$WT(m,n,k,P) \leq \tfrac{m \times n}{P} \times (1+\tfrac{P^2-P}{R \times C}) \times (\tfrac{k}{k-1})^2. \tag{27}$$

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



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 13: The Three Phases of a Parallel Fill Cache Subproblem

FastLSA algorithm (i.e., Equation 6), it can be inferred that

$$WT(m,n,k,P) = PFillCacheT(m,n,k,P) + (2k-1) \times WT(\tfrac{m}{k}, \tfrac{n}{k}, k, P). \tag{28}$$

The first step of the proof is to find a good approximation for $PFillCacheT(M,N,k,P)$. As explained in Section 5.1, the DPM 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 Hirschberg, we have $T = O(\frac{M \times N}{R \times C})$.

As shown in Figure 7, the computation of the tiles advances following a diagonal wavefront pattern. In Figure 7, 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

13 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 $(P - 1)T$.

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 13. 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 13).

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.,

$$(R \times C - u \times v) - \frac{P(P-1)}{2} - (\frac{P(P-1)}{2} - u \times v) = R \times C - P^2 + P. \tag{29}$$

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

$$\frac{(R\times C-P^2+P)}{P}\times T. \tag{30}$$

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

$$\begin{aligned} PFillCacheT(M,N,k,P) &= (P-1)T+\frac{(R\times C-P^2+P)}{P}T+(P-1)T \\ &= \frac{(R\times C+P^2-P)}{P}T \\ &= \frac{(R\times C+P^2-P)\times M\times N}{P\times R\times C} \\ &= M\times N\times\frac{1}{P}(1+\frac{P^2-P}{R\times C}) \\ &= M\times N\times\alpha, \end{aligned} \tag{31}$$

where

$$\alpha=\frac{1}{P}(1+\frac{P^2-P}{R\times C}). \tag{32}$$

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

$$\begin{aligned} PBaseCaseT(M,N,P) &= (P-1)T + (R\times C - \tfrac{P(P-1)}{2} - \tfrac{P(P-1)}{2})\tfrac{T}{P} + (P-1)T \\ &= (P-1)T + \tfrac{(R\times C - P^2 + P)}{P}T + (P-1)T \\ &= M \times N \times \tfrac{1}{P}(1 + \tfrac{P^2-P}{R\times C}) \\ &= M \times N \times \alpha. \end{aligned} \tag{33}$$

Using the results of Equation 31 and Equation 33, Formula 28 becomes

$$\begin{aligned} WT(m,n,k,P) &= m\times n\times\alpha + (2k-1)\times WT(\tfrac{m}{k}, \tfrac{n}{k}, k, P) \\ &= mn\alpha + (2k-1)(\tfrac{m}{k}\tfrac{n}{k}\alpha + (2k-1)WT(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k, P)) \\ &= mn\alpha + mn\alpha\tfrac{2k-1}{k^2} + (2k-1)^2 WT(\tfrac{m}{k^2}, \tfrac{n}{k^2}, k, P) \\ &= mn\alpha + mn\alpha\tfrac{2k-1}{k^2} + mn\alpha(\tfrac{2k-1}{k^2})^2 + (2k-1)^3 WT(\tfrac{m}{k^3}, \tfrac{n}{k^3}, k, P) \\ &= \cdots = \\ &= mn\alpha(1 + \tfrac{2k-1}{k^2} + (\tfrac{2k-1}{k^2})^2 + \cdots + (\tfrac{2k-1}{k^2})^{a-1}) + (2k-1)^a WT(\tfrac{m}{k^a}, \tfrac{n}{k^a}, k, P) \\ &= mn\alpha(1 + \tfrac{2k-1}{k^2} + \cdots + (\tfrac{2k-1}{k^2})^{a-1}) + (2k-1)^a PBaseCaseT(\tfrac{m}{k^a}, \tfrac{n}{k^a}, P) \\ &= mn\alpha(1 + \tfrac{2k-1}{k^2} + \cdots + (\tfrac{2k-1}{k^2})^{a-1}) + (2k-1)^a \tfrac{m}{k^a}\tfrac{n}{k^a}\alpha \\ &= mn\alpha(1 + \tfrac{2k-1}{k^2} + \cdots + (\tfrac{2k-1}{k^2})^{a-1} + (\tfrac{2k-1}{k^2})^a) \\ &= mn\alpha\tfrac{1-(\frac{2k-1}{k^2})^{a+1}}{1-\frac{2k-1}{k^2}}. \end{aligned} \tag{34}$$

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

$$\begin{aligned} WT(m,n,k,P) &= mn\alpha\tfrac{1-(\frac{2k-1}{k^2})^{a+1}}{1-\frac{2k-1}{k^2}} \\ &\leq mn\alpha\tfrac{1}{1-\frac{2k-1}{k^2}} = mn\alpha(\tfrac{k}{k-1})^2. \end{aligned} \tag{35}$$

By replacing α with its value (Equation 32), it becomes true that

$$WT(m,n,k,P) \leq mn\alpha(\frac{k}{k-1})^2 = \frac{m\times n}{P} \times (1 + \frac{P^2-P}{R\times C}) \times (\frac{k}{k-1})^2, \tag{36}$$

which concludes the proof of Theorem 4. ■