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

Adrian Driga, Paul Lu, Jonathan Schaeffer, Duane Szafron, Kevin Charter, and Ian Parsons
Dept. of Computing Science, University of Alberta
Edmonton, Alberta, T6G 2E8, Canada
{adrian|paullu|jonathan|duane}@cs.ualberta.ca

Abstract

Pairwise 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 amount of available memory. Our experiments show that, in practice, due to memory caching effects, FastLSA is always as fast or faster than Hirschberg and the FM algorithms. We have also parallelized FastLSA using a simple but effective form of wavefront parallelism. Our experimental results show that Parallel FastLSA exhibits good speedups.

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, 17, 7, 14], 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 globally-optimal pairwise sequence alignment. Given FastLSA's strong analytical and empirical characteristics with respect to space and time complexity, FastLSA is a good candidate for parallelization.

As the second contribution, we show that FastLSA is nicely parallelizable while maintaining the strong complexity properties of the sequential algorithm.

The third contribution is an empirical study of Parallel FastLSA and a discussion of 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.

2 Background and Related Work

The primary structure of a protein consists of a sequence of amino acids, 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 possible. 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
* * * * *        * *  ** *
```

A scoring function (e.g., the Dayhoff scoring matrix, MDM78 Mutation Data Matrix - 1978 [5]) is used to evaluate and choose among the different possible alignments. Exact matches (e.g., `D` aligned with `D`) are given high scores (assuming that high scores are desired) and inexact matches (e.g., `K` aligned with `V`) are given low scores. If an amino acid in one sequence lines up with a gap in the other sequence (e.g., `K` aligned with `-`), 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 [13] and Smith and Waterman [18]. Aligning two sequences of length m and n is equivalent to finding the maximum cost path through a dynamic programming matrix (DPM) of size $m + 1$ by $n + 1$, where an extra row and column is added to capture leading gaps. 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. Given that we now have the capacity to sequence entire genomes, pairwise sequence comparisons involving up to four million nucleotides at a time are now desirable. O($m \times n$) storage of this magnitude would require memory sizes beyond the range of current technology.

Hirschberg [10] was the first to report a linear space algorithm. 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 tradeoff: 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 [12] applied it to sequence alignment.

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

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.

In the area of pairwise sequence alignment, *BLAST* (Basic Local Alignment Search Tool) [1], is currently the most commonly-used tool. In contrast to FastLSA, BLAST does not attempt to find the globally-optimal alignment. There is significant biological motivation for locally-optimal alignments, as with BLAST, but globally-optimal alignments, as with FastLSA, are still interesting and useful.

In the area of parallel algorithms, Aluru, Futamura, and Mehrotra [2] suggest an embarrassingly parallel algorithm for sequence alignment, which they refer to as the *Parallel Space-Saving* algorithm, a generalization of Hirschberg's algorithm. The Parallel Space-Saving algorithm builds on the ideas of Edmiston *et al.* [9]. The drawback of this parallel algorithm is the lack of control on the granularity of the subproblems it generates. To achieve good speedups, the subproblems should have similar sizes. This is unlikely to happen in practice because of the irregular nature of the biological sequences to be aligned. As we will see, FastLSA also has granularity issues, but it also has parameters that can be tuned to deal with granularity.

```
Algorithm FastLSA       /* Will parallelize parallelFastLSA() in Section 4 */
    input : logical-d.p.-matrix flsaProblem,
            cached-values cacheRow and cacheColumn,
            solution-path flsaPath
    output: optimal path corresponding to flsaProblem prepended to flsaPath

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

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

    /* Figure 2 (c).  Can parallelize as parallelFillGridCache() */
5   fillGridCache( flsaProblem, flsaGrid )

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

    /* Figure 2 (d).  Recursion. */
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 2 (e) .  Recursion. */
13      flsaPathExt = FastLSA( flsaSubProblem, newCacheRow, newCacheColumn,
                               flsaPathExt )

14  deallocateGrid( flsaGrid )

    /* Figure 2 (f) */
15  return flsaPathExt
```

Figure 1. Pseudo-Code for FastLSA

Martins *et al.* [11] have a parallel version of the Needleman-Wunsch algorithm. The DPM is divided into equally-sized blocks, and the algorithm statically preassigns rows of blocks to each processor. This algorithm suffers from the same major drawback as the original Needleman-Wunsch algorithm: the space required is quadratic in the size of the sequences. The particular implementation considered is based on EARTH, "a fine-grain event-driven multi-threaded execution and architecture model" [11]. The performance numbers presented, although impressive, are obtained through simulation, and the largest DPM computed for their benchmarks has only $4,000 \times 10,000$ entries. We present empirical results on parallel hardware and our problem sizes are substantially larger.

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, via parameter k, to take advantage of different cache memory and main memory

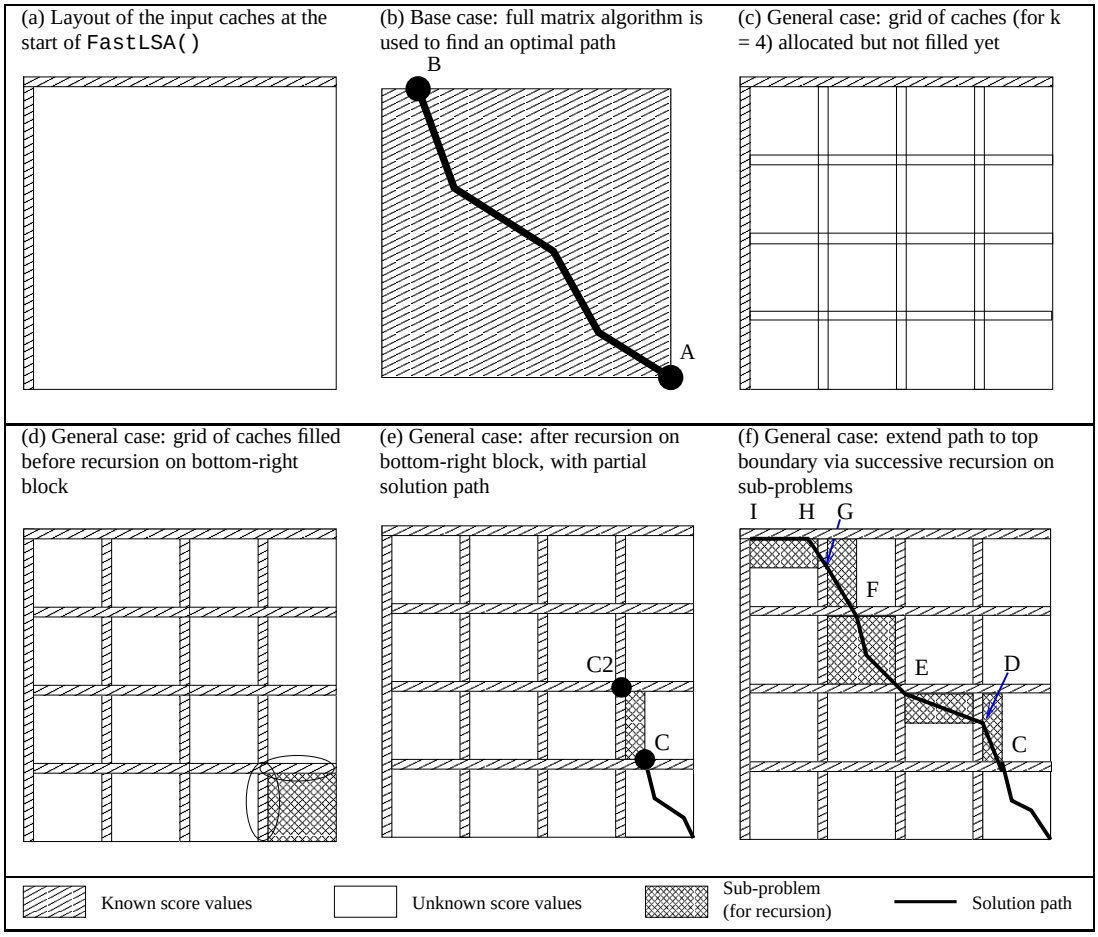

Figure 2. Execution Stages of FastLSA

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 DPM in *grid cache* 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 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 1 and an explanatory diagram is 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.

Prior to running FastLSA, BM units of memory are reserved from the RM units available. These reserved units are referred to as the *Base Case buffer*. If the DPM 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 1).

The full matrix algorithm uses the input values `cacheRow` and `cacheColumn` as the first row and column of the DPM it must compute (Figure 2 (a)). After all entries of the DPM have been computed, an optimal path through the matrix is built. Figure 2 (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 bigger than BM, the General Case of the algorithm is followed (Figure 1). In this case, FastLSA splits the input problem into smaller subproblems. These subproblems are solved recursively. The solution paths for these subproblems, if concatenated, form a solution path for the input problem.

It is useful to observe that FastLSA solves a succession of rectangular problems, called *FastLSA subproblems*, as either a Base Case for small subproblems, or as a Fill Cache for subproblems that do not fit in the Base Case buffer (Figure 2(c) to Figure 2(f)).

Let $S(m, n, k)$ be the maximum number of DPM entries that need to be stored in order to align the sequences using a grid cache of k rows and k columns; $k - 1$ rows of length n and $k - 1$ columns of length m must be allocated. $S(m, n, k) \leq k \times (m + n) + BM$ [4, 8], so we know that the space overhead is linear with respect to problem size. Admittedly, FastLSA uses more space than Hirschberg's algorithm, but FastLSA also recomputes fewer DPM entries, thus improving the overall performance. Furthermore, FastLSA is conveniently parameterized by k and can be adjusted to use all RM units of memory.

Again, FastLSA trades space for time. 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. The total execution time of FastLSA is proportional to $T(m, n, k)$. In the worst case, $T(m, n, k) = m \times n \times \frac{k+1}{k-1}$ for FastLSA [4]. 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.

We compared the empirical performance of the FM algorithm, Hirschberg's algorithm, and FastLSA using a common software and hardware base. 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 (memory) bus, 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 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 [3] to serve as our query sequences. The average and standard deviation of the real times for the 5 queries are in Table 1. Note that, with one exception, FastLSA is the fastest algorithm.

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 × 100 × 4 bytes = 40 Kbytes to 100 × 5000 × 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. Hirschberg's algorithm also fits into the secondary cache. However, since it does more re-computations than FastLSA, it cannot overtake the FM algorithm.

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

Table 1. Sequential Search of the Swiss-Prot Databases with FM, Hirschberg and FastLSA (times in $seconds \times 10^3$, fastest times are in boldface)

From Table 1 we conclude that for shorter sequences, the choice of the best algorithm depends on cache effects. However, FastLSA is always better than Hirschberg's algorithm. For longer sequences, FastLSA is the best choice.

4 Parallel FastLSA

To further improve performance, sequential FastLSA can, in theory, be parallelized via two major components:

1. Base Case: the full matrix algorithm used for solving Base Case subproblems (line 2 of the pseudo-code from Figure 1), and
2. General Case: the computation of the FastLSA Grid Cache for the Fill Cache subproblems (line 5 of the pseudo-code from Figure 1).

The only changes from the sequential version to Parallel FastLSA are the replacement of the sequen-

tial `solveFullMatrix`() with a parallel version, `parallelSolveFullMatrix`(), in line 2, and the replacement of the sequential `fillGridCache`() with a parallel version, `parallelFillGridCache`(), in line 5.

In practice, we found that the Base Case subproblems are already too fine-grained to benefit from parallelism. Therefore, in the following section, we analyze the performance of an implementation of Parallel FastLSA that solves all Base Case subproblems sequentially; the Fill Cache subproblems of the General Case are the only ones solved in parallel.

As discussed earlier, parameter k is the primary control of the storage overhead of FastLSA. However, new parameters u and v, where $R = u \times k$ and $C = v \times k$ (Figure 3) are introduced to control the parallel work partitioning strategy for the Fill Cache subproblems. Too few units of work results in too many idle processors; too many units of work results in poor speedups due to fine-grained work.

Each Fill Cache subproblem is subdivided into *tiles*, which are laid out along R rows and C columns. Note that each tile contains many DPM entries. In Figure 3, $u = 2$, $v = 3$, $k = 4$ and (consequently) $R = 8$, $C = 12$; the actual parameter values are selected to tune the performance of the algorithm (e.g., the different values of u and v in Table 2). At any moment during the parallel computation, 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.

In terms of the order in which work is computed, the computation starts with one processor computing the entries of the top-left tile (labelled `1` in Figure 3). The computation of the top-left tile is possible because the initial row (i.e., `cacheRow`) and column values (i.e., `cacheColumn`) for this tile are available. In fact, the top-left tile is the only tile that has all its initial values available. All the other processors are idle during Step 1.

After Step 1, 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. In Step 2, the two tiles neighboring the top-left tile, labelled `2` in Figure 3, can be computed in parallel on two different processors.

The processing of the tiles advances on a diagonal-like front. In Figure 3, 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 $(k^2 - 1) \times u \times v$ tiles have been computed. The empty region at the bottom-right of the Fill Cache subproblem is solved by recursion.

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 ready tiles 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 statically preassigned to the processors, and each tile is processed as soon as it becomes ready. The performance results for Parallel FastLSA presented in this section are obtained using an implementation based on the dynamic distribution of work strategy.

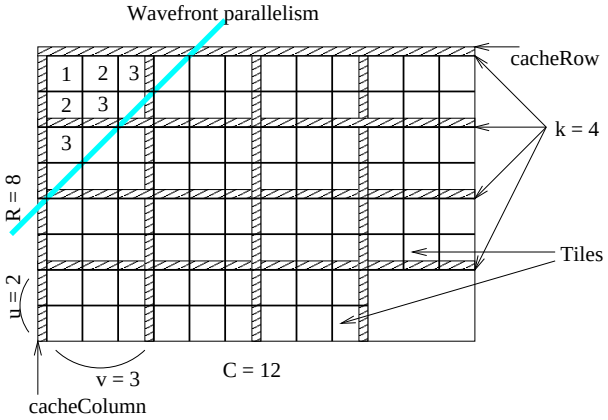

Figure 3. Data Partitioning for Parallel Fill Cache Subproblems

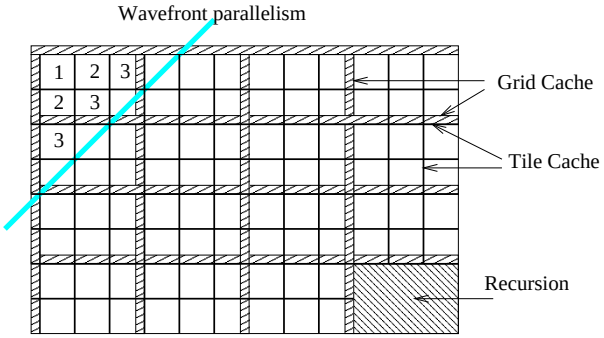

Figure 4. FastLSA Grid Cache and Tile Cache for Parallel Fill Cache Subproblems

In terms of data storage, the Tile Cache (Figure 4) is needed to hold the intermediate results passed between tiles. For example, Tile Caches hold the right-most column and the bottom-most row of the top-left tile (labelled `1`); the information is used in computing the tiles labelled `2`.

Figure 4 shows the Grid Cache delimiting the submatrices and, in turn, the Tile Cache delimiting the tiles. 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. Note that new caches of each type, FastLSA Grid Cache and Tile Cache, are allocated in shared memory for each Fill Cache subproblem solved.

4.1 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. The full derivation and proofs of the following are elsewhere [8].

4.1.1 FastLSA Recursion Pattern

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:

$$\begin{aligned} &PFastLSA(m, n, k) = \\ &\quad PFillCache(m, n, k), PFastLSA(\tfrac{m}{k}, \tfrac{n}{k}, k), \\ &\quad PFastLSA(m_1, n_1, k), \ldots, PFastLSA(m_z, n_z, k); \end{aligned} \quad (1)$$

where $PFillCache(m, n, k)$ is the initial Fill Cache subproblem, $PFastLSA(\frac{m}{k}, \frac{n}{k}, k)$ is the recursive call to the bottom-right subproblem, and $PFastLSA(m_i, n_i, k)$, $i = 1, z$ are the subproblems solved recursively inside the `while`-loop of the algorithm. Depending on the configuration of the optimal alignment path that is followed by the FastLSA algorithm, z can take values between $k - 1$ and $2k - 2$. Details about the values of z in the best case and worst case scenarios can be found in [4].

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

$$\tfrac{m}{k^a} \times \tfrac{n}{k^a} \leq BM < \tfrac{m}{k^{a-1}} \times \tfrac{n}{k^{a-1}}. \quad (2)$$

This is equivalent to

$$a - 1 < \tfrac{\log \frac{m \times n}{BM}}{2 \log k} \leq a \Leftrightarrow \left\lceil \tfrac{\log \frac{m \times n}{BM}}{2 \log k} \right\rceil = a. \quad (3)$$

4.1.2 Space Complexity

Definition 1 *Let $S(m, n, k)$ be the maximum number of DPM entries that need to be stored 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 1 *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*

$$S(m, n, k) \leq (3k-1) \times (m+n) + \tfrac{P}{C} \times n + R \times C - u \times v + BM. \quad (4)$$

4.1.3 Time Complexity

Definition 2 *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 2 *Let $WT(m, n, k, P)$ be defined as in Definition 2. 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*

$$WT(m, n, k, P) \leq \tfrac{m \times n}{P} \times (1 + \tfrac{P^2 - P}{R \times C}) \times (\tfrac{k}{k-1})^2. \quad (5)$$

Therefore, the algorithm is still quadratic in its time complexity.

5 Experimental Results for Parallel FastLSA

To establish that Parallel FastLSA can achieve reasonable speedups, in practice, for globally-optimal pairwise alignment, we present results from experiments 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 Mbytes 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]. Our scoring function, which is simpler than the Dayhoff matrix, assigns identical matches a score of 2, all mismatches a score of -1, and a gap penalty of -2.

We discuss 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 [14]. Most of their examples are comparisons of "some region of the human genome with the synthetic region from a rodent genome" [16]. These pairs are used as a test suite, not only because of their size, but also because their alignment is biologically meaningful to the Penn State group.

Name	Value	Notes
Constants		
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;
Variables		
P	1, 2, 4, 8, 16, 32	number of processors;
k	8 to 12	number of Grid rows and columns;
R	$u \times k = 3 \times k$	total number of rows of tiles for a Fill Cache subproblem;
C	$v \times k = 4 \times k$	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 2. FastLSA Parameters

Admittedly, there is a valid debate as to the length of the sequences that biologists actually wish to align. Towards that, as computing scientists, we can only say that we have been motivated by biologists who do seem to want to do large alignments [6, 17, 7].

We have experimented with several more pairs of DNA sequences, but 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 [16]. The human sequence is 37,785 base pairs (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*) [16] 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" [15]. The human sequence is 319,030 bp long, and the mouse sequence is 305,636 bp long.

Several tunable parameters introduced in Section 4 are assigned constant, empirical values in this study (Table 2). The parameters most relevant to parallel processing are left are variables. Constraining some of the parameters is justified since we are primarily interested in establishing reasonable performance for Parallel FastLSA rather than optimal performance. In the future, we hope to further explore the parameter space. Table 2 summarizes the parameters involved in the FastLSA algorithms and the values assigned to them.

Note that 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.

To remove the small, unpredictable noise generated by the operating system, three consecutive runs are performed for each set of parameter values. The three time samples obtained for each run are averaged.

5.1 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 an 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 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.

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 result-

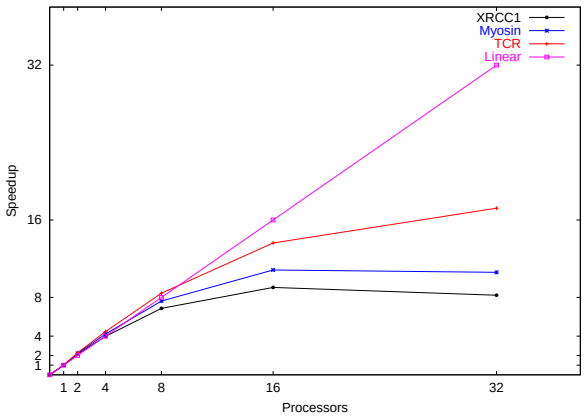

Figure 5. Best Speedups for XRCC1, Myosin, and TCR

ing speedup curves are shown in Figure 5. Table 3 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.

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

Sequences	Processors	Time (s)	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 3. Real Times, Speedups, and k

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.

Not surprisingly, the best speedup curve is obtained for the largest sequences that are aligned; our empirical results show that Parallel FastLSA can scale with the problem size. 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.

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.

5.2 Subproblem Types and Sizes in the Myosin Dataset

The time spent by the FastLSA algorithms in computing a pairwise alignment is dominated by the total time spent by the algorithms on filling matrices for Base Case subproblems or filling Grid Caches for Fill Cache subproblems. We can trace and cluster the subproblems based on the type and size of the subproblem.

A subproblem count graph (Figure 6) shows how many FastLSA subproblems are solved during an alignment operation and how coarse-grained are the problems. Naturally, coarse-grained subproblems are more-easily exploited for parallel computation than fine-grained subproblems. Note that the FastLSA subproblems which occur for an alignment

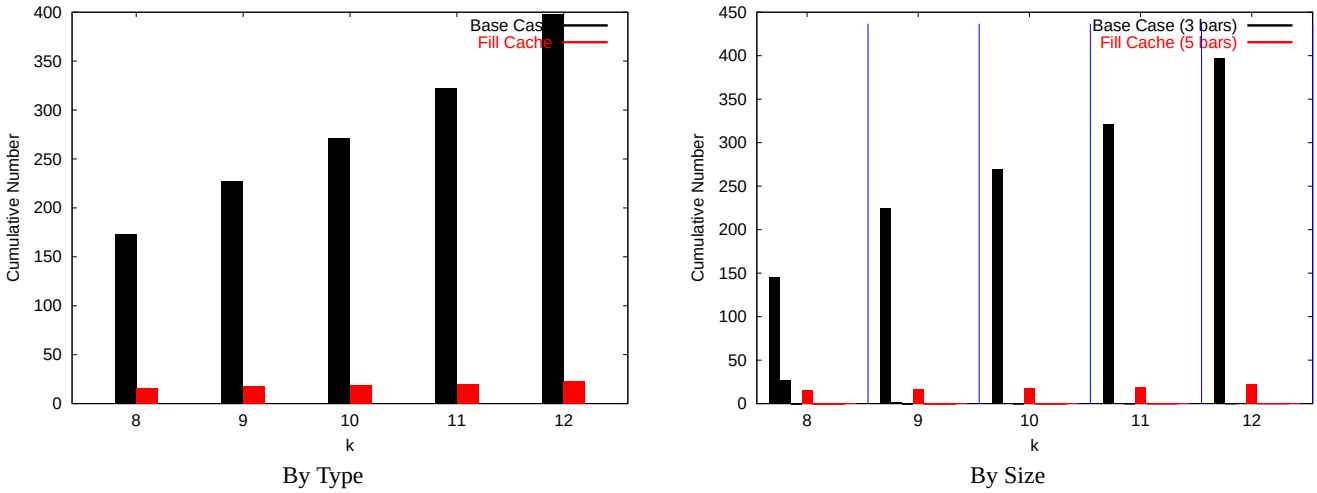

Figure 6. FastLSA Subproblem Count: Parallel FastLSA Alignment for Human *Myosin* versus Hamster *Myosin* (Breakdown Based on the Type/Size of the FastLSA Subproblems)

are determined by the specific sequences themselves (e.g., Myosin), the size of the Base Case buffer (BM), and k, but are independent of the number of processors used for the alignment. Figure 6 consists of two parts: (LHS, By Type) one for the clustering based on the type of the subproblems and (RHS, By Size) the other for the clustering based on the size of the subproblems.

According to the "By Type" (Figure 6) graph, Base Case subproblems (black bars; left-most bar for each k) dominate Fill Cache subproblems (red bars; right-most bar for each k) in terms of the number of subproblem instances (i.e., cumulative number count).

In the "By Size" graph, the black bars (left-most bars for each k) show the distribution of Base Case problems across three different sizes: 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. Since the largest black bars are on the far left of each group in the 'By Size" graph, we conclude that most of the Base Case problems are $\frac{1}{3}BM$ or smaller in size. Normally, the most common type of work or subproblem is a good candidate for parallelization, but the Base Case subproblems are low-granularity computations and, therefore, are best computed sequentially, as per our previous design decision.

For Fill Cache subproblems, the right-most red bars of the 'By Size" graph, the interval between BM and the size of the initial DPM is evenly divided into five subintervals. Most of the Fill Cache subproblems are small relative to the full DPM, but they are larger than BM (by definition) and have sufficient granularity to generate speedups.

6 Concluding Remarks

Sequence alignment is a fundamental operation for homology search in bioinformatics. The Fast Linear Space Alignment (FastLSA) algorithm adapts to the amount of space available by trading space for time. 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 reasonable speedups for up to 16 processors, with problems of sufficient size; 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. Our empirical results are a first look at the large parameter space, with more future work indicated. However, notably, existing algorithms for sequence alignment (i.e., FM and Hirschberg) are not similarly parameterized. Given the large DNA sequences (e.g., tens of thousands of bases) that some researchers wish to study [6, 17, 7], the space and time complexity of a sequence alignment algorithm become increasingly important. The combination of FastLSA's parameterized storage complexity, good analyt-

ical time complexity, easy parallelization, and reasonable empirical performance makes FastLSA a good choice for globally-optimal pairwise sequence alignment.

7 Acknowledgments

We would like to acknowledge Scott Fortin at BioTools (`www.biotools.com`) 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 National Science and Engineering Research Council (NSERC), the Alberta Informatics Circle of Research Excellence (iCORE) and the Canada Foundation for Innovation (CFI).

References

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

[2] S. Aluru, N. Futamura, and K. Mehrotra. Parallel biological sequence comparison using prefix computations. In *International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, pages 467–473, April 1999.

[3] 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/.

[4] K. Charter, J. Schaeffer, and D. Szafron. Sequence alignment using FastLSA. In *International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS)*, pages 239–245, 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] E.W. Edmiston, N.G. Core, J.H. Saltz, and R.M. Smith. Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming*, 17(3):259–275, June 1988.

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

[11] 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.

[12] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences (CABIOS)*, 4:11–17, 1988.

[13] 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.

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

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

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

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

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