# Computer Science Technical Report

**Analysis and Performance of a UPC Implementation of a Parallel Longest Common Subsequence Algorithm**

by
Bryan Franklin and Steven Seidel

Computer Science Technical Report
CS-TR-09-01

November 30, 2009


**Department of Computer Science**
**Houghton, MI 49931-1295**
**www.cs.mtu.edu**

# Contents

# List of Figures

**Abstract**

An important problem in computational biology is finding the longest common subsequence (LCS) of two nucleotide sequences. This paper examines the correctness and performance of a recently proposed parallel LCS algorithm that uses successor tables and pruning rules to construct a list of sets from which an LCS can be easily reconstructed. Counterexamples are given for two pruning rules that were given with the original algorithm. Because of these errors, performance measurements originally reported cannot be validated. The work presented here shows that speedup can be reliably achieved by an implementation in Unified Parallel C that runs on an Infiniband cluster. This performance is partly facilitated by exploiting the software cache of the MuPC runtime system. In addition, this implementation achieved speedup without bulk memory copy operations and the associated programming complexity of message passing.

# 1 Introduction

## 1.1 The Longest Common Subsequence Problem

Given a sequence of symbols $X = x_1x_2x_3...x_n$ from an alphabet $\Sigma$, a *subsequence* is any sequence obtained by deleting zero or more symbols from $X$. A sequence that is a subsequence of each member of a collection of sequences is called a *common subsequence*. The *k-LCS problem* is to find at least one longest common subsequence of a set of $k$ sequences. The best known instances of this problem are in computational biology where the goal is to determine the similarity of DNA sequences or protein sequences [8, 2, 23, 14]. In the case of DNA, the sequences are strings over the alphabet $\Sigma = \{a, c, g, t\}$. For amino acids the alphabet consists of 20 symbols [14]. While $k$-LCS is NP-complete [17], it is solvable in polynomial time using dynamic programming when the number of sequences is fixed [27]. Only LCS problems where $k = 2$ are considered in this paper, however a brief discussion of how pLCS might can be modified to solve the k-LCS problem is given in Section 2.4.

## 1.2 Edit Distance Problem

A related problem to LCS is the edit distance problem. Given two sequences $X$ and $Y$ of symbols from an alphabet $\Sigma$, the edit distance problem is to find a minimum cost sequence of weighted edit operations that are applied to $X$ to transform it into $Y$. Valid edit operations are delete a single symbol, insert a single symbol or replace a single symbol with a different symbol [3]. The longest common subsequence problem is a special case of the edit distance problem, in which the cost of an insertion or deletion is 1, the cost of replacing a character with a different character is 2, and the cost of leaving a character unchanged is 0 [27].

## 1.3 Background

Many serial algorithms have been put forth for solving the LCS problem and the edit distance problem (which LCS is a special case of). Bergroth *et al.* [4] compared many such algorithms. The widely used Smith-Waterman algorithm [25] for the edit distance is based on the work of Needleman and Wunsch [22]. Hirschberg showed that LCS can be solved in $O(n \cdot m)$ using $O(n + m)$ space [12] using a recursive divide and conquer approach. Ullman, Aho and Hirschberg [26] determined the lower bound on time-complexity of LCS

to be $\Omega(mn)$ using the decision tree model, where $m$ and $n$ are the lengths of the input sequences. Masek [18] and Myers [19] gave four-russians algorithms with run-times of $O(n^2/\log n)$. Using suffix tree methods for finding the LCS of gene sequences MUMmer [6] and MGA [13] algorithms are both fast and memory efficient [7]. Guo [11] gives a linear space primal-dual algorithm that runs in $O(nL)$ time where $L$ is the length of the LCS.

The longest common subsequence problem has also been solved using systolic array algorithms. Robert and Tchuente [24] gave an algorithm that solves LCS in $n + 5m$ steps using $m(m + 1)$ processing elements. Chang *et al.* [5] gave an algorithm that runs in $4n + 2m$ steps using $m(m + 1)$ processing elements. Luce *et al.* [16] gave an algorithm that runs in $n + 3m + L$ steps using $m(m + 1)/2$ processing elements. Freschi and Bogliolo [9] gave an algorithm that find the LCS of two run-length-encoded (RLE) sequences in $O(m + n)$ steps using $M + N$ processing elements.

Several parallel algorithms for LCS have also been proposed. Myoupo and Semé [20] gave an algorithm that runs on the broadcasting with selective reduction (BSR) model which claims a constant time solution to the LCS problem. For the CREW-PRAM model Aggarwal and Park [1] and Apostolico *et al.* [3] gave algorithms that run in $O(\log \log n)$ time on $mn \log m$ processors. Lu and Lin [15] gave two algorithms for LCS, one that runs in $O(\log^2 n + \log m)$ time on $mn/\log m$ processors, and one that runs in $O(\log^2 m + \log \log m)$ time on $mn/\log^2 m \log \log m$ processors when $\log^2 m \log \log m > \log n$, otherwise it takes $O(\log n)$ time on $mn/\log n$ processors. Apostolico *et al.* [3] gave an $O(\log n\,(\log \log m)^2)$ algorithm that runs on $O(mn/\log \log m)$ processors. Babu and Saxena [21] converted existing CREW-PRAM algorithms to the CRCW-PRAM model giving run times of $O(\log^2 n)$ using $mn$ processors.

Several parallel algorithms have also be proposed for solving the edit distance problem. Edmiston *et al.* [8] gave parallel versions of both Needleman-Wunsch and Smith-Waterman algorithms. Galper and Brutlag also gave a parallel Smith-Waterman algorithm based on dynamic programming [10]. Zhang *et al.* [28] proposed a parallel version of the Smith-Waterman algorithm that uses a divide and conquer strategy to reduce memory usage.

Liu *et al.* proposed FAST_LCS, a parallel LCS algorithm [14] and reported it to have a run time of $O(L)$, where $L$ is the length of an LCS, using $O(n + m + P)$ space, where $P$ is the number of identical pairs (see Definition 2). The parallel LCS algorithm, ***pLCS***, presented here is based on FAST_LCS. This paper notes and corrects errors in FAST_LCS as it was given in [14]. It is shown that pLCS has a runtime of $O(n^3/T)$ on $T$ processors and uses $O((n/T)^2)$ space per processor, where $n = \min\{n, m\}$. The UPC implementation of pLCS described here exploits spatial locality to take advantage of the software cache of the MuPC runtime system [29]. This implementation exhibits speedup without the use of any

explicit bulk shared memory copy operations.

## 1.4 Dynamic Programming Algorithm

One common serial approach to solving the the LCS problem is by dynamic programming [12]. Solving the LCS problem by dynamic programming is done in two phases. In the first phase an $n+1$ by $m+1$ matrix is computed. In the second phase the actual LCS can easily be reconstructed using the original strings and the matrix.

### 1.4.1 Computing LCS Matrix

Given two input strings $X = x_1x_2...x_n$ with $n$ symbols and $Y = y_1y_2...y_m$ with $m$ symbols, an $n+1$ by $m+1$ matrix $M$ is constructed according to the recurrence given in Figure 1.

$$M_{i,j} = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ 1 + M_{i-1,j-1} & x_i = y_j \\ max(M_{i-1,j}, M_{i,j-1}) & \text{otherwise} \end{cases}$$

Figure 1: Recurrence for computing Longest Common Subsequence (LCS) Matrix.

Since the computation of each cell in the matrix depends on the values of the three cells above, to the left of and diagonally above and to the left, this matrix can be filled in a top down, left to right fashion.

### 1.4.2 Reconstructing LCS

Let $L$ initially be the empty string. Starting from $M_{i,j}$, whenever $x_i = y_j$, move to prepend $x_i$ to $L$ and move to $M_{i-1,j-1}$. Otherwise move to the larger of $M_{i-1,j}$ and $M_{i,j-1}$. When the cell $M_{0,0}$ is reached, $L$ will contain an LCS of $X$ and $Y$.

### 1.4.3 Time and Space Complexity

Since this dynamic programming algorithm must fill the matrix, which takes $O(n \cdot m)$ time, then reconstruct the LCS which take $O(n + m)$ time, the overall time complexity is $O(n \cdot m)$. Also since the entire $n + 1$ by $m + 1$ matrix is typically stored, along with the original input strings, the space complexity is $O(n \cdot m)$. However [12] showed that LCS can be solved in $O(n + m)$ space and $O(n \cdot m)$ time using a recursive divide and conquer algorithm.

## 1.5 Parallel Dynamic Programming

Several parallel techniques have been proposed to perform dynamic programming similar to that used in solving LCS. While the methods listed here are intended for solving the edit-distance problem, the same techniques, can be applied to filling the matrix used to solve the LCS problem.

Edmiston *et al.* [8] used the message passing paradigm, and subdivided the matrix into rectangular sub-matrices, then each processor filled a sub-matrix. Once each sub-matrix was filled, the final row was passed to another processor so it could use it to start another sub-matrix.

Galper and Brutlag [10] on the other hand used the shared-memory paradigm and decomposed the computations into wavefronts in a variety of ways. The first was the row wavefront method where each processor attempted to fill in a single row of the matrix. Because of the data dependencies in the recurrence, a row can only be filled from left to right up to the point of the row above it. An alternative was a diagonal wavefront, in which each processor started at the left edge of the matrix and moved up along an anti-diagonal. Using the row wavefront method gives a run-time in $O(m \cdot n/p)$ where $m$ is the length of the query string, $n$ is the size of the database when run on $p$ processors. The diagonal wavefront method gives a run-time in $O(m^2/p)$.

# 2 The pLCS Algorithm

## 2.1 The algorithm

This section describes the basis of the pLCS algorithm as it was given by Liu, *et al.* [14]. The idea behind the algorithm is to follow chains of pairs of identical symbols, one from each string, from which an LCS can be constructed. Each iteration of the algorithm adds one link to each of the chains currently under consideration. Chains that provably cannot be part of an LCS are pruned after each iteration. Each iteration is called a "level". The number of levels equals the length of an LCS. When the computation of the last level is complete, an LCS can be constructed by a backwards traversal of any chain that has persisted to the last level.

**Definition 1:** Given a string (or *sequence*) $X = x_1x_2...x_n$ over an alphabet $\Sigma$ of $S$ symbols, the *successor table* of $X$ is an $S \times (n+1)$ table $T_X$ such that each entry $T_X(\sigma, i)$, where $0 \leq i \leq n$ and $\sigma \in \Sigma$, gives the position of the next instance of character $\sigma$ after position $i$ in $X$. When there are no more instances of a symbol $\sigma$ after position $i$, $T_X(\sigma, i) =$–.

For example, the successor tables for $T_X$ for $X = tactacgc$ and $T_Y$ for $Y = gtcgaag$ are shown in Figures 2 and 3.

**Definition 2:** Given two sequences $X$ and $Y$ and their successor tables $T_X$ and $T_Y$, an *identical pair* is a tuple $(i, j, (p, q)) = (T_X(\sigma, p), T_Y(\sigma, q), (p, q))$, where $\sigma \in \Sigma$. Note that

| $T_X$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| a | 2 | 2 | 5 | 5 | 5 | – | – | – | – |
| c | 3 | 3 | 3 | 6 | 6 | 6 | 8 | 8 | – |
| g | 7 | 7 | 7 | 7 | 7 | 7 | 7 | – | – |
| t | 1 | 4 | 4 | 4 | – | – | – | – | – |

Figure 2: Successor table for the sequence *tactacgc*.

| $T_Y$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| a | 5 | 5 | 5 | 5 | 5 | 6 | – | – |
| c | 3 | 3 | 3 | – | – | – | – | – |
| g | 1 | 4 | 4 | 4 | 7 | 7 | 7 | – |
| t | 2 | 2 | – | – | – | – | – | – |

Figure 3: Successor table for the sequence *gtcgaag*.

this implies $x_i = y_j$ and, when none of the indices are –, $p < i$ and $q < j$. For simplicity, the third component $(p, q)$ of an identical pair is often omitted and the object is referred to as just a *pair*. The pair $(i, j)$ is said to be a *successor* of the pair $(p, q)$, and $(p, q)$ is a *predecessor* of $(i, j)$.

**Definition 3:** The *initial pairs* are the identical pairs of the form $(T_X(\sigma, 0), T_Y(\sigma, 0), (0, 0))$, for each $\sigma \in \Sigma$.

**Definition 4:** $level_1$ is the set of initial pairs. For $k > 1$, $level_k$ is the set of pairs that are the successors of pairs in $level_{k-1}$.

For example, in Figures 2 and 3, the initial pairs for $a$, $c$, $g$, and $t$ are $(2, 5)$, $(3, 3)$, $(7, 1)$, and $(1, 2)$, respectively, and $(5, 6)$ and $(7, 7)$ are successors of $(2, 5)$. The first 4 levels for this instance of the problem are nonempty, as shown in Figure 4.

The algorithm terminates when it finds that $level_{k+1}$ is empty. The reverse of an LCS can then be constructed starting from any pair $(i, j, (p, q))$ in $level_k$ and tracing backward to any of its predecessors $(p, q, ...)$ in $level_{k-1}$, and so on, back to a pair of the form $(r, s, (0, 0))$ in $level_1$.

For example, $level_5$ in Figure 4 is empty, so the longest common subsequence of *tactacgc* and *gtcgaag* has a length of 4. One of the longest common subsequences is given in reverse by the pairs $(7, 7, (5, 6)) \rightarrow (5, 6, (2, 5)) \rightarrow (2, 5, (1, 2)) \rightarrow (1, 2, (0, 0))$, which corresponds to $x_1x_2x_5x_7 = y_2y_5y_6y_7 = taag$.

The correctness of pLCS follows from the definitions. Any LCS must begin with at least one symbol from an initial pair, otherwise a longer common subsequence could be constructed. Moreover, some LCS must begin with symbol $x_i = y_j$, where $(i, j, (0, 0))$ is an initial pair. Every pair of consecutive symbols in an LCS corresponds to pairs that are

| Level | Pairs |
|---|---|
| 1 | { (2,5,(0,0)), (3,3,(0,0)), (7,1,(0,0)), (1,2,(0,0)) } |
| 2 | { (5,6,(2,5)), (7,7,(2,5)), (5,5,(3,3)), (7,4,(3,3)), (8,3,(7,1)), (2,5,(1,2)), (3,3,(1,2)), (7,4,(1,2)) } |
| 3 | { (7,7,(5,6)), (7,7,(5,5)), (5,6,(2,5)), (7,7,(2,5)), (5,5,(3,3)), (7,4,(3,3)) } |
| 4 | { (7,7,(5,6)), (7,7,(5,5)) } |
| 5 | $\emptyset$ |

Figure 4: Trace of pLCS algorithm on the sequences *tactacgc* and *gtcgaag*.

successors of one another. That is, if $x_i = y_j$ and $x_{i+i'} = y_{j+j'}$ are consecutive symbols in an LCS, then $(i + i', j + j')$ is a successor of $(i, j)$. The algorithm traces all chains of successors and so each of the longest chains the algorithm finds must be an LCS.

Algorithm 1 is pseudo code for the sequential version of pLCS.

**Algorithm 1** pLCS

```
1: Build successor tables
2: // Get initial pairs
3: for all σ ∈ Σ do
4:    (T_X(σ,0), T_Y(σ,0), (0,0)) ∈ level_1
5: end for
6: // Find all relevant identical pairs
7: k ← 1
8: repeat
9:    // Apply pruning rules to level_k
10:   for all (i,j) ∈ level_k do
11:      for all (p,q) ∈ level_k s.t. (p,q) ≠ (i,j) do
12:         if (i,j) can be pruned due to (p,q) then
13:            Prune (i,j)
14:         end if
15:      end for
16:   end for
17:   // Get successor pairs
18:   for all (i,j) ∈ level_k and σ ∈ Σ do
19:      Put (T_X(σ,i), T_Y(σ,j), (i,j)) in level_{k+1}
20:   end for
21:   k ← k + 1
22: until level_k = ∅
23: // Reconstruct LCS
24: (i,j) ∈ level_{k-1}
25: Z ← λ
26: repeat
27:   Z ← x_i Z
28:   (i,j) ← predecessor of (i,j)
29: until (i,j) is an initial pair
```

## 2.2 Pruning Rules

The following theorems were given in Liu, *et al.* [14]. These theorems show that pairs that satisfy certain properties can be deleted (*pruned*) from the level at which they occur. In these theorems, let $X = x_1...x_n$ and $Y = y_1...y_m$. Notation of the form $X_{pq}$ denotes the substring $x_p x_{p+1}...x_q$ of $X$. Similarly for $Y$.

**Theorem 1:** If a level contains two pairs $(i, j)$ and $(k, l)$, where $i > k$ and $j > l$, then the pair $(i, j)$ can be pruned.

**Proof:** All LCSes of $X_{in}$ and $Y_{jm}$ are strictly shorter than LCSes of $X_{kn}$ and $Y_{lm}$ because $X_{in}$ and $Y_{jm}$ are proper suffixes of $X_{kn}$ and $Y_{lm}$, respectively, and the latter two sequences have an identical pair $(k, l)$ not in the former sequences. It follows that $(i, j)$ can be pruned from the current level. □

The following theorem shows that in the special case that $j = l$, the pair $(i, j)$ can also be pruned.

**Theorem 2:** If a level contains two pairs $(i, j)$ and $(k, j)$ where $i > k$, then the pair $(i, j)$ can be pruned.

**Proof:** $X_{in}$ is a proper suffix of $X_{kn}$, so no LCS of $X_{in}$ and $Y_{jm}$ can be longer than an LCS of $X_{kn}$ and $Y_{jm}$, so $(i, j)$ can be pruned. □

**Corollary 1:** If a level contains two pairs $(i, j)$ and $(i, k)$ where $j > k$, then pair $(i, j)$ can be pruned.

As in the example shown in Section 2.1, let $X = tactacgc$ and $Y = gtcgaag$. Figure 5 gives a trace of pLCS with pruning rules applied.

In this example pairs (2,5) and (3,3) in $level_1$ can be pruned due to the pair (1,2) by Theorem 1. In $level_2$ the pair (7,4) can be pruned due to (3,3) by Theorem 1, while (8,3) can be pruned due to (3,3) by Theorem 2. In $level_3$, the pair (7,7) can be pruned due to (5,6) by Theorem 1, while (5,6) can be pruned due to (5,5) by Corollary 1.

Since $level_5$ in Figure 5 is empty, the longest common subsequence of *tactacgc* and *gtcgaag* has a length of 4. The longest common subsequence that was found is given in reverse by the pairs $(7, 7, (5, 5)) \rightarrow (5, 5, (3, 3)) \rightarrow (3, 3, (1, 2)) \rightarrow (1, 2, (0, 0))$, which corresponds to $x_1 x_3 x_5 x_7 = y_2 y_3 y_5 y_7 = tcag$.

| Level | Pairs |
|---|---|
| 1 | { **(2,5,(0,0))**, **(3,3,(0,0))**, (7,1,(0,0)), (1,2,(0,0)) } |
| 2 | { **(8,3,(7,1))**, (2,5,(1,2)), (3,3,(1,2)), **(7,4,(1,2))** } |
| 3 | { **(5,6,(2,5))**, **(7,7,(2,5))**, (5,5,(3,3)), (7,4,(3,3)) } |
| 4 | { (7,7,(5,5)) } |
| 5 | $\emptyset$ |

Figure 5: Trace of pLCS algorithm on the sequences *tactacgc* and *gtcgaag* with pruning. Pruned pairs are indicated in bold.

## 2.3 Counterexamples

The following two claims were made in [14]. We provide counterexamples to those claims.

**Claim 1:** If a level contains two pairs $(i, j)$ and $(i + 1, k)$, where $k < j$, then the pair $(i, j)$ can be pruned.

**Counterexample:** Let $X = gaga$ and $Y = aaga$. The successor tables for these strings are given in Figures 6 and 7. As Figure 8 shows, the last nonempty level is 3, so the LCS has length 3. However when Claim 1 is applied to $level_2$, the pair (3,3) is pruned due to (4,2) since $4 = 3 + 1$ and $2 < 3$. This causes the algorithm to finish after $level_2$, due to (4,2) and (2,4) not having any successors, giving a longest common subsequence of 2.

| $T_X$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 2 | 2 | 4 | 4 | – |
| c | – | – | – | – | – |
| g | 1 | 3 | 3 | – | – |
| t | – | – | – | – | – |

Figure 6: Successor table for the sequence *gaga*.

| $T_Y$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 1 | 2 | 4 | 4 | – |
| c | – | – | – | – | – |
| g | 3 | 3 | 3 | – | – |
| t | – | – | – | – | – |

Figure 7: Successor table for the sequence *aaga*.

| Level | Pairs |
|---|---|
| 1 | { (2,1,(0,0)), (1,3,(0,0)) } |
| 2 | { (4,2,(2,1)), (3,3,(2,1)), (2,4,(1,3)) } |
| 3 | { (4,4,(3,3)) } |
| 4 | $\emptyset$ |

Figure 8: Trace of pLCS algorithm on the sequences *gaga* and *aaga*, without any pruning.

| $T_X$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| a | 2 | 2 | 5 | 5 | 5 | – | – |
| c | – | – | – | – | – | – | – |
| g | 1 | 3 | 3 | – | – | – | – |
| t | 4 | 4 | 4 | 4 | 6 | 6 | – |

Figure 9: Successor table for the sequence *gagtat*.

| Level | Pairs |
|---|---|
| 1 | { (2,1,(0,0)), (1,3,(0,0)) } |
| 2 | { (5,2,(2,1)), (3,3,(2,1)), (2,4,(1,3)) } |
| 3 | { (5,4,(3,3)) } |
| 4 | $\emptyset$ |

Figure 10: Trace of pLCS algorithm on the sequences *gagtat* and *aaga*, without any pruning.

**Claim 2:** If a level contains two pairs $(i, j)$ and $(i + 2, k)$, where $k < j$ and $x_{i+1} = x_{i+3}$, then the pair $(i, j)$ can be pruned.

**Counterexample:** Let $X = gagtat$ and $Y = aaga$. The successor tables for these strings are given in Figures 9 and 7. As Figure 10 shows, the last nonempty level is 3, so the LCS has length 3. However when Claim 2 is applied to $level_2$, the pair (3,3) is pruned due to (5,2) since $5 = 3 + 2$, $2 < 3$ and $x_4 = x_6 = t$. This causes the algorithm to finish after $level_2$, due to (5,2) and (2,4) not having any successors, giving a longest common subsequence of 2.

## 2.4 Modifications needed for *k*-LCS

The pLCS algorithm while designed to solve the LCS problem for two input strings. It can be modified to solve the more general $k$-LCS problem [14]. These changes include increasing the size of the pair structure to be a $k$-tuple structure. The pruning rules would also need slight revision.

Given two $k$-tuples $t_0 = (i_0, i_1, \ldots, i_k)$ and $t_1 = (j_0, j_1, \ldots, j_k)$ within the same level, $t_0$ can be pruned if $t_1 \neq t_2$ and $i_p \geq j_p$ for all $p$ s.t. $0 \leq p \leq k$, . As in the 2-LCS case, when a level contains duplicate $k$-tuples, only one of them should be kept.

## 2.5 Rebalancing

Since the time required to prune a level in pLCS is proportional to the product of the sizes of the two largest per thread pair lists, for best performance all threads should have the same number of pairs. However due to pruning, some threads will have more pairs than others. To correct this, a greedy rebalancing algorithm was implemented. Pseudocode for this algorithm is given as Algorithm 2.

The rebalancing algorithm first has one thread gather the number of pairs from all $k$ threads. Threads are denoted $T_i$. The number of pairs in thread $T_i$ is denoted $n_i$. The number of pairs for each thread is then scattered to all threads. Each thread then sorts the list of pair counts to get a new list of threads $T'$, where each $T'_i$ has $n'_i$ pairs. Using the sorted list of threads, each thread computes a $k$ by $k$ matrix $steal$ that gives how many pairs $T_i$ needs to steal from thread $T_j$, like the one shown in Figure 11. Each thread also has a count of how many pairs it will be keeping, $keep_i$. Initially, $keep_i$ is the minimum of the target value and the number of pairs the thread has.

Before the $steal$ matrix can be computed, the target number of pairs needs to be computed. This value is $\overline{n'}$, which may not be an integer. When the target value is not an integer, threads that have more than the target number of pairs are allowed to keep extra pairs, and reduce the number of pairs that are available to be stolen. This reduction starts with the thread that has the most pairs, and moves one pair from $n'_i$ into $keep_i$, it then moves down

| | | Receiver | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ |
| Sender | $T_0$ | 23 | – | – | – | – | – | – | – |
| | $T_1$ | – | 24 | – | – | – | 3 | 1 | – |
| | $T_2$ | – | – | 24 | – | – | – | – | 7 |
| | $T_3$ | 1 | – | – | 24 | – | – | 1 | – |
| | $T_4$ | – | – | – | – | 24 | 2 | – | 2 |
| | $T_5$ | – | – | – | – | – | 18 | – | – |
| | $T_6$ | – | – | – | – | – | – | 22 | – |
| | $T_7$ | – | – | – | – | – | – | – | 14 |

Figure 11: Matrix computed using algorithm shown in Algorithm 2.

through the threads until $\overline{n'}$ is an integer. If a thread is reached that has too few pairs, the process continues from the thread that has the most pairs.

Once $n'$ has been adjusted, $steal$ can be computed. The $steal$ matrix is initialized such that $steal_{i,i} \leftarrow \min(n'_i, \overline{n'})$ The computation then starts by computing the number of pairs that $T'_0$ needs, $need_i \leftarrow \overline{n'} - n'_i$, and how many pairs $T'_k$ has available, $avail_j \leftarrow n'_j - \overline{n'}$. If $avail_j > need_i$, then $steal_{i,j} \leftarrow need_i$ and $i \leftarrow i + 1$. If $avail_j < need_i$, then $steal_{i,j} \leftarrow avail_j$ and $j \leftarrow j - 1$. If $avail_j = need_i$, then $steal_{i,j} \leftarrow avail_j$, $i \leftarrow i + 1$ and $j \leftarrow j - 1$. This process continues so long as $j > i$. By computing $steal_{i,j}$ in this way, thread $T'_i$ always tries to get pairs from the thread that is likely to have enough.

Using the $\textit{steal}_{i,j}$ matrix, each thread can compute the starting offset of a block of pairs to be stolen by finding how many other threads with a smaller thread id are also stealing from that threads. Thus $\textit{offset}_{i,j} = \sum_{m=0}^{j-1} \textit{steal}_{i,m}$. At this point thread $T'_i$ knows that it needs to steal pairs at positions $\textit{offset}_{i,j}$ through $\textit{offset}_{i,j}+\textit{steal}_{i,j}$ from thread $T'_j$'.

Since the computation of the matrix is the same on all threads, all threads will all agree on which pairs every other thread will be stealing. Once each thread knows which pairs it will be stealing from other threads, each thread copies the needed pairs by simply reading the pairs out of the remote thread's shared memory.

By stealing pairs in this fashion, a single contiguous block of pairs will always be stolen from one thread by another. Which means the thread that needs pairs can steal the pairs directly from shared memory one pair at a time making efficient use of the cache due to spatial locality.

It should be noted that tests were performed to verify that it is faster to have all threads compute the communication needed, than to have one thread gather the counts, compute and scatter the communication matrix.

**Algorithm 2** Rebalance

Gather then scatter per thread pair counts
Sort threads by pair counts
Compute average number of pairs per thread

// Reduce pair counts
$keep_i \leftarrow \min(n_i, \lfloor \overline{n} \rfloor), \quad \forall i$
**while** $\overline{n'} > \lfloor \overline{n'} \rfloor$ **do**
  $j \leftarrow \text{THREADS} - 1$
  **while** $\overline{n'} > \lfloor \overline{n'} \rfloor$ **do**
    $n'_j \leftarrow n'_j - 1$
    $keep_j \leftarrow keep_j + 1$
  **end while**
**end while**

// Build matrix
$i \leftarrow 0$
$j \leftarrow \text{THREADS} - 1$
**while** $i < j$ **do**
  $needed_i = \overline{n'} - n_i$
  **while** $needed_i > 0$ **do**
    $allowed_j = n'_j - \overline{n'}$
    $steal_{i,j} = \min(allowed_j, needed_i)$
    $needed_i \leftarrow needed_i - steal_{i,j}$
    $n'_j \leftarrow n'_j - steal_{i,j}$
    **if** $allowed_j = 0$ **then**
      $j \leftarrow j - 1$
    **end if**
  **end while**
  $i \leftarrow i + 1$
**end while**

// Copy pairs as needed
$i \leftarrow \text{MYTHREAD}$
**for** $j = 0$ to THREADS-1 **do**
  *offset* $\leftarrow \sum_{k=0}^{j-1} steal_{k,j}$
  $count \leftarrow steal_{i,j}$
  Steal $steal_{i,j}$ pairs starting at position *offset* from thread $T'_j$.
**end for**

### 2.5.1 Rebalancing Example

This section provides an example of the rebalancing algorithm used in pLCS. At the start of the algorithm each thread ($T_i$) has a number of pairs ($n_i$), as shown in Figure 12. In the first phase of rebalancing thread $T_0$ gathers the pair counts from each thread, then each scatters the data to all threads. Each thread then computes the average number of pairs per thread ($\overline{n}$), which is will be the goal number of pairs after rebalancing.

Once all threads have a copy of the pair counts table, they each sort their local copies. After sorting, the pair counts of some threads are reduced slightly to simplify rebalancing when $\overline{n}$ is not an integer. Starting from the thread that has the most pairs, 1 is subtracted from its pair count, then the next thread has one subtracted, so forth and so on until the total number of pairs is a multiple of the number of threads. If a thread is reached that has a pair count less than or equal to $\lfloor \overline{n} \rfloor$, the process is started over from the thread that has the largest pair count. This results in pair counts that sum to a multiple of THREADS, as shown in Figure 13.

Let $T'_i$ be the thread with the $i$th smallest number of pairs. also Let $n'_i$ be the number of pairs in thread $T'_i$ after being reduced.

After the pair count tables are sorted and reduced so that $\overline{n'}$ is an integer, a matrix is computed indicating how many pairs are to be stolen by each thread and which thread they will be stolen from as shown in Figure 11. Figure 15 also gives the number of pairs to be stolen, but the columns and rows have been re-ordered according to each threads initial number of pairs, giving a clearer view of the communication pattern. The starting offset for the block of pairs that will be stolen is given in the matrix shown in Figure 14. Using these two matrices, the actual communication operations can be determined, as Figure 16 shows.

| Thread | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | Total | $\lfloor \overline{n} \rfloor$ | Remainder |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $n_i$ | 23 | 28 | 31 | 26 | 28 | 18 | 22 | 14 | 190 | 23 | 6 |

Figure 12: Original pair counts for each thread.

| | $T'_0$ | $T'_1$ | $T'_2$ | $T'_3$ | $T'_4$ | $T'_5$ | $T'_6$ | $T'_7$ |
|---|---|---|---|---|---|---|---|---|
| Thread | $T_7$ | $T_5$ | $T_6$ | $T_0$ | $T_3$ | $T_1$ | $T_4$ | $T_2$ |
| $n_i$ | 14 | 18 | 22 | 23 | 26 | 28 | 28 | 31 |
| Reduced by | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 2 |
| $n'_i$ | 14 | 18 | 22 | 23 | 25 | 27 | 26 | 29 |

Figure 13: Threads sorted by pair counts with pair counts reduced so their sum is divisible by THREADS.

| | | Receiver | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ |
| Sender | $T_0$ | 0 | – | – | – | – | – | – | – |
| | $T_1$ | – | 0 | – | – | – | 0 | 0 | – |
| | $T_2$ | – | – | 0 | – | – | – | – | 0 |
| | $T_3$ | 23 | – | – | 0 | – | – | 1 | – |
| | $T_4$ | – | – | – | – | 0 | 3 | – | 7 |
| | $T_5$ | – | – | – | – | – | 5 | – | – |
| | $T_6$ | – | – | – | – | – | – | 2 | – |
| | $T_7$ | – | – | – | – | – | – | – | 9 |

Figure 14: Offset matrix computed using algorithm shown in Algorithm 2.

| | | Receiver | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $T'_0$ | $T'_1$ | $T'_2$ | $T'_3$ | $T'_4$ | $T'_5$ | $T'_6$ | $T'_7$ |
| Sender | $T'_0$ | 14 | – | – | – | – | – | – | – |
| | $T'_1$ | – | 18 | – | – | – | – | – | – |
| | $T'_2$ | – | – | 22 | – | – | – | – | – |
| | $T'_3$ | – | – | – | 23 | – | – | – | – |
| | $T'_4$ | – | – | 1 | 1 | 24 | – | – | – |
| | $T'_5$ | – | 3 | 1 | – | – | 24 | – | – |
| | $T'_6$ | 2 | 2 | – | – | – | – | 24 | – |
| | $T'_7$ | 7 | – | – | – | – | – | – | 24 |

Figure 15: Matrix shown in Figure 11 in $T'$ order.

After applying the rebalancing actions, the final number of pairs in each thread is given in Figure 17. Clearly the minimum and maximum number of pairs only differ by one, which is as close to balanced as possible.

### 2.5.2 Number of messages

This algorithm works in a greedy fashion and does not maintain the original ordering of pairs within a level. However it does manage to rebalance in $O(\text{THREADS})$ messages.

**Theorem:** The total number of pair blocks sent between threads is always less than the number of threads.
**Proof:** Let $G = (V, E)$ be a directed graph that represents the communication needed for rebalancing, where each element $T_i$ of $V$ represents a thread, and $(T_i, T_j) \in E$ indicates that thread $T_j$ steals a contiguous block of pairs from thread $T_i$. Assuming rebalancing is actually needed, some threads must have fewer pairs than other threads, and the mean

| Thread | Keeps | Steals |
|---|---|---|
| $T_0$ | 23 | $Pairs_3[25]$ |
| $T_1$ | 24 | none |
| $T_2$ | 24 | none |
| $T_3$ | 24 | none |
| $T_4$ | 24 | none |
| $T_5$ | 18 | $Pairs_1[1 \ldots 3]$<br>$Pairs_4[2 \ldots 3]$ |
| $T_6$ | 22 | $Pairs_1[0]$<br>$Pairs_3[0]$ |
| $T_7$ | 14 | $Pairs_2[0 \ldots 6]$<br>$Pairs_4[0 \ldots 1]$ |

Figure 16: Communication operations needed to rebalance.

| Thread | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ |
|---|---|---|---|---|---|---|---|---|
| Pairs | 24 | 24 | 24 | 24 | 24 | 23 | 24 | 23 |

Figure 17: Final pair counts for each thread after rebalancing.

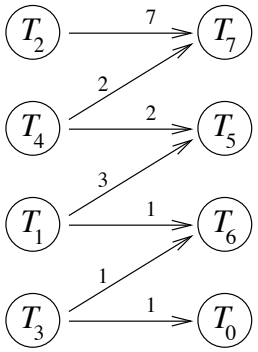


Figure 18: Bipartite graph showing communication patterns for example given in Section 2.5.1. Edges are labeled with the number of pairs moving between threads.

number of pairs per thread must be between the minimum and maximum number of pairs. Thus the threads can be divided into two groups, one that has more than the average number of pairs (i.e. the threads that will be stolen from) and one that has fewer pairs (i.e. the threads that will be doing the stealing). If these two groups are then arranged into two columns where the left column contains the threads that have above average pair counts ordered largest to smallest, and the right column contains threads with below average thread counts ordered smallest to largest, as shown in Figure 18.

As the rebalance algorithm runs, it will first add an edge between the thread with the most pairs, and the thread with the least pairs, which are the top nodes in each column. In each

subsequent iteration the head of the next edge will be moved if the stealing thread has enough pairs, and the tail will move if the source thread is out of extra pairs. Since one of these conditions will always be true, at least one end of each edge will be moved down to the next node each iteration. The maximum number of edges is $|V| - 1$ since the first first iteration starts by selecting two nodes, and each subsequent iteration selects one more node to add an edge to. □

### 2.5.3 Rounds of communications

In this rebalancing algorithm there is an initial phase of gathering and scattering the pair counts for each thread which can be viewed as two rounds of communications. Once the pair counts are scattered to all threads, each thread is able to compute which pairs will be stolen from other threads. Then another set of communication rounds starts, the total number of rounds is given by the maximum in-degree or out-degree of the nodes in the communication graph (Figure 18) which is given by treating Figure 11 as an adjacency matrix.

### 2.5.4 Rebalance Complexity

In order to rebalance, the pair counts for $T$ threads must first be gathered then scattered to all threads. This can be done in roughly $O(\lg T)$ time. The thread counts are then sorted, which takes $O(T \cdot \lg T)$ time. Computing the communication matrix takes $O(T)$ time. Each thread can then compute the portion of the offset matrix that affects it in $O(T)$ time. Thus the overall time complexity of computing the communication needed is $O(T \cdot \lg T)$.

Once the necessary communication is computed, the moving of pairs can be done using at most $T - 1$ messages, as shown in Section 2.5.2. This can therefore be done in $O(T)$ rounds. Thus the overall complexity for rebalancing is $O(T \cdot \lg T)$ time.

The space complexity for rebalancing as described is $O(T^2)$ due to the $T$ by $T$ matrix that is computed, however storing values in this matrix is not necessary as the stealing of pairs can occur as soon as the number of pairs and starting offset are computed. So, only the pairs counts for each thread need to be stored, thus the overall space complexity for rebalancing is $O(T)$.

# 3 Crossing Pairs

In pLCS performance can be suffer when a level contains a large number of *crossing pairs*, which are any two pairs $(i, j)$ and $(k, l)$ such that $i < k$ and $j > l$.

The upper portion of Figure 19 shows the crossing pairs $(i, j)$ and $(k, j)$ from the strings $agtacgt$ and $gcatgca$, where $(i, j)$ refers to the first $t$ in each string and $(k, j)$ refers to the first $c$ in each string. The lower portion of Figure 19 shows the possible relationships two pairs $(i, j)$ and $(k, l)$ can have depending on the relative values $i, j, k$ and $l$ values. In the case of separated pairs, one of the pairs will be pruned by Theorem 1. In the case of touching pairs, one of the pairs will be pruned by Theorem 2 or Corollary 1. This section discusses attempts to prune crossing pairs, and gives counterexamples that demonstrate problems with each method attempted.

## 3.1 Pruning of Crossing Pairs

In [14], two pruning rules were given for which counterexamples are given in Sections 2.3 and 2.3, which attempt to prune crossing pairs.

Since crossing pairs are common in pLCS, it would be useful to prune as many of them as possible. However as this section will demonstrate, it is difficult to reliably and efficiently

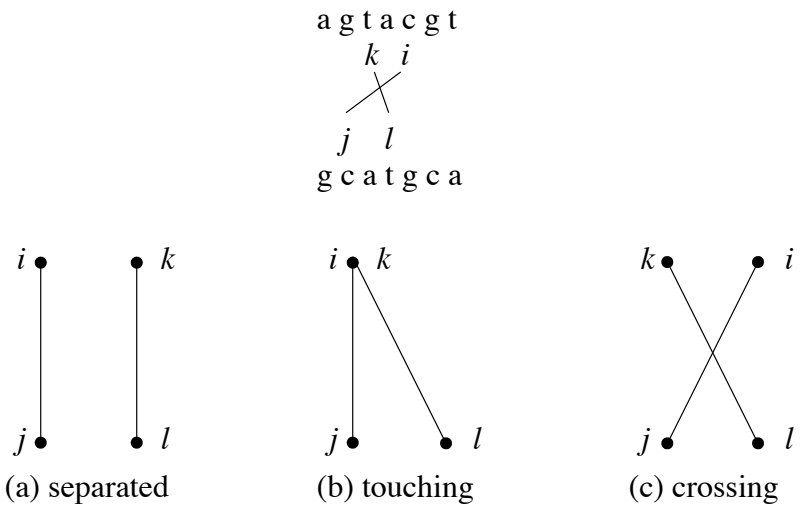


Figure 19: Possible relationships between two pairs $(i, j)$ and $(k, l)$: a) separated, b) touching and c) crossing.

identify pruneable crossing pairs.

Starting with the assumption that the claimed pruning rules in [14] contained only minor flaws, several attempts were made to create valid pruning rules from them.

**Theorem 3:** When a level contains two pairs $(i, j)$ and $(i + r, k)$ where $k < j$, $x_{i+1} \ldots x_{i+r-1} = x_{i+r+1} \ldots x_{i+2r-1}$ and $r > \max_{\sigma \in \Sigma} (T_X(\sigma, i) - i)$ (i.e. $r$ is large enough that the substring $x_{i+1} \ldots x_{i+r-1}$ contains every character that can be found in the substring $x_{i+1} \ldots x_n$), then pair $(i, j)$ can not always be pruned without affecting the length of the LCS.

**Proof:** By counterexample. Assume that pairs meeting the requirements for $(i, j)$ of Theorem 3 are always pruneable without affecting the length of the longest common subsequence found. Let $X = aacaaa$ and $Y = caacaa$. The successor tables for these strings are given in Figures 20 and 21. Using the corresponding successor tables shown in Figures 20 and 21, the levels shown in Figure 22 can be computed. Which gives an LCS of length 5.

However, if pairs that match the criteria for $(i, j)$ in Theorem 3 are allowed to be pruned, the pair $(2, 3)$ in $level_2$ would be pruned due to the pair $(4, 2)$. This pruning would then lead to a common subsequence with a length of only 4. □

Several modifications to the pruning rule in Theorem 3 were also tested. The first modification added the constraint that $x_{i+1} \ldots x_{i+r-1}$ must contain at least one instance of each character in $\Sigma$. Another attempt modified the sub-strings that are matched to be $x_i \ldots x_{i+r-1}$

| $T_X$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| a | 1 | 2 | 4 | 4 | 5 | 6 | – |
| c | 3 | 3 | 3 | – | – | – | – |
| g | – | – | – | – | – | – | – |
| t | – | – | – | – | – | – | – |

Figure 20: Successor table for the sequence *aacaaa*.

| $T_Y$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| a | 2 | 2 | 3 | 5 | 5 | 6 | – |
| c | 1 | 4 | 4 | – | – | – | – |
| g | – | – | – | – | – | – | – |
| t | – | – | – | – | – | – | – |

Figure 21: Successor table for the sequence *caacaa*.

and $x_{i+r} \ldots x_{i+2r-1}$. Finally the constraint that length($X_{kn}$) > length($Y_{jm}$) was tested. All three of these modified pruning rules were tested and shown to cause imprecise results for some input strings. While all of these modifications were tried separately, it is believed that combinations would also lead to imprecise results.

Since Theorems 1 and 2 give pruning rules that reduce the pairs to a point where each pair must have a unique $i$ value and a unique $j$ value. An attempt was made to develop a pruning rule based on restricting pairs to unique diagonals.

**Theorem 4:** If a level contains two pairs $(i, j)$ and $(k, l)$ such that $i+j = k+l$, and $|i - j| > |k - l|$, then it is not always possible to prune $(i, j)$ without affecting the correctness of pLCS.

**Proof:** Assume that if a level contains two pairs $(i, j)$ and $(k, l)$, where $(i + j = k + l)$, that $(i, j)$ can be pruned. Let $X = gaca$ and $Y = ca$. The successor tables for these strings are given in Figures 23 and 24.

Since the last non-empty level in Figure 25 is $level_2$, the length of the LCS is 2 However if pairs $(i, j)$ matching the criteria of Theorem 4 were pruned, then $(3, 1)$ in $level_1$ could be pruned. Since $(3, 1)$ is the predecessor of $(4, 2)$, $level_2$ would be empty, giving and LCS length of 1. □

| Level | Pairs |
|---|---|
| 1 | { (1,2,(0,0)), (3,1,(0,0)) } |
| 2 | { (2,3,(1,2)), (3,4,(1,2), (4,2,(3,1)) } |
| 3 | { (4,5,(2,3)), (3,4,(4,5,(3,4)), (4,5,(3,4)), (5,3,(4,2)) } |
| 4 | { (5,6,(4,5)), (4,5,(3,4)), (6,5,(5,3)) } |
| 5 | { (5,6,(4,5)) } |
| 6 | ∅ |

Figure 22: Trace of algorithm on the sequences *aacaaa* and *caacaa*, without any pruning.

| $T_X$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a | 2 | 2 | 4 | 4 | – |
| c | 3 | 3 | 3 | – | – |
| g | 1 | – | – | – | – |
| t | – | – | – | – | – |

Figure 23: Successor table for the sequence *gaca*.

# 4 Complexity Analysis

Let $X = x_1..x_n$ and $Y = y_1..y_m$ be an instance of the LCS problem. W.l.o.g., we may assume that $n = \min\{n, m\}$. Also, $S = |\Sigma|$ is assumed to be constant.

## 4.1 Sequential run time

The algorithm generates at most $n$ nonempty levels because the LCS can not be longer than the shorter input sequence. $level_1$ has exactly $S$ pairs. Generating a successor pair is $O(1)$ because it is a simple table look-up. $level_{k+1}$ has at most $S$ times the number of pairs in $level_k$, but this exponential growth is strictly limited by pruning. Theorems 1 and 2 and Corollary 1 guarantee that at most $n$ pairs remain after pruning. Treating pairs as Cartesian coordinates in an $n \times m$ array, Theorem 1 says there can be no pairs $(x, y)$ in the rectangular region $x > i$, $y > j$. Similarly, Theorem 2 and Corollary 1 say that there can be at most one pair per $y$ and $x$ coordinate, respectively. Pruning $n$ pairs has cost $O(n^2)$. This gives an overall run time bound of $O(n^3)$.

The memory required to run pLCS is $\Theta(m)$ for the successors tables, and $O(n^2)$ for the pairs in the worst case.

While this complexity is higher than the $O(nm)$ worst case run time bound on the dynamic programming algorithm [12], parallelizing the pLCS algorithm offered many interesting implementation challenges. It proved to be a good example of the performance benefits of

| $T_Y$ | 0 | 1 | 2 |
|---|---|---|---|
| a | 2 | 2 | – |
| c | 1 | – | – |
| g | – | – | – |
| t | – | – | – |

Figure 24: Successor table for the sequence *ca*.

| Level | Pairs |
|---|---|
| 1 | { (2,2,(0,0), (3,1,(0,0)) } |
| 2 | { (4,2,(3,1)) } |
| 3 | $\emptyset$ |

Figure 25: Trace of algorithm on the sequences *gaca* and *ca*, without any pruning.

a runtime cache as well as a demonstration of a complex application in which the cache can be used by the programmer to avoid explicit message passing.

## 4.2 Parallel run time

As in the serial algorithm there are at most $n$ levels and each level has at most $O(n)$ pairs after pruning. The pairs are load balanced by an exchange of pairs between threads that have too many and those that have too few. Computing how many pairs and which pairs need to be moved between threads takes $O(T \cdot \lg T)$ time as discussed in section 2.5.4. This is done in $O(T)$ rounds of communication, where $T$ is the number of threads (or processors), and it ensures that at the beginning of each iteration each thread has $O(n/T)$ pairs. Generating a successor pair has constant cost, as in the serial algorithm, because each thread has complete copies of the two successor tables. (Distributing the successor tables caused too many remote references.) Applying the pruning rules to a level is done in $T$ rounds of communication. $O((n/T)^2)$ pair comparisons are done on each round. The run time of the parallel algorithm thus consists of $O(n)$ levels, each with $O(T)$ rounds of communication with $O((n/T)^2)$ comparisons of pairs on each round. This yields a total cost of $O(nT(n/T)^2) = O(n^3/T)$. This run-time is significantly larger than the $O(n^2/T)$ for parallel dynamic programming, however further study would be needed to see if exploiting the data caching in MuPC can lead to better performance on certain problem sizes.

Since each thread contains a copy of the two successor tables, the per-thread memory bound for pLCS is $O(m + n^2/T)$. For large problem sizes where $n$ and $m$ are comparable, $n^2/T$ will dominate.

# 5 pLCS Implementation in UPC

We implemented the pLCS algorithm in UPC. This implementation was designed to be run on the MuPC runtime system [29]. It takes affinity and the availability of a runtime cache into account, so it is not a “naive" implementation. In particular, during pruning all references to remote pairs are local once the pairs are cached. All necessary references are made to a cached pair before going on to the next pair. This maximizes cache reuse. In addition, since all remote pruning operations are done to pairs in cache, all pairs in the cache are evaluated for pruneability before being written back. Thus pairs are only ever read or written along with an entire cache line.

One shortcoming of MuPC is that it only checks the cache for remote objects that are no more than 16 bytes long. The implementation represents a pair as two integers and a pointer to shared, so a pair is larger than 16-bytes. Care was taken to ensure that the components of a pair were always referenced individually, and not the pair structure as a whole, so that the reference would be resolved in the cache and not cause a separate remote communication operation in the runtime system.

## 5.1 Data Structures

In order to run pLCS two main data structures are needed. First there is the successor table, which is used to generate new identical pairs in each level. The second data structure holds the actual levels and the pairs within them.

### 5.1.1 Successor Tables

Since all threads will need to access both successor tables, the successor tables are declared as shared. However, for ease of implementation and performance reasons the shared successor tables are computed by one thread, then each thread makes a private copy of the successor tables which will be used in all subsequent computations.

The structure of a successor table is an array of $n + 1$ successor structures, where $n$ is the length of the input string it is built from. The successor structure in turn contains a single array of $|\Sigma|$ integers which indicate the next position of a particular symbol in the input string.

### 5.1.2 Pairs and Levels

During the operation of the pLCS algorithm a series of levels are produced where each level contains a set of pairs. Each pair consists of two locations in the input strings, a pruned flag and a pointer to the pair's predecessor. The positions are stored as integers. To save space, the sign bit for one of the position indexes is used for the pruned flag. Since the pointer to the predecessor is only needed when reconstructing the LCS, in order to pack more pair structures in a single cache line, the predecessor pointers are stored in a separate array of predecessor structures. The predecessor structure contains two pointers, the first one points

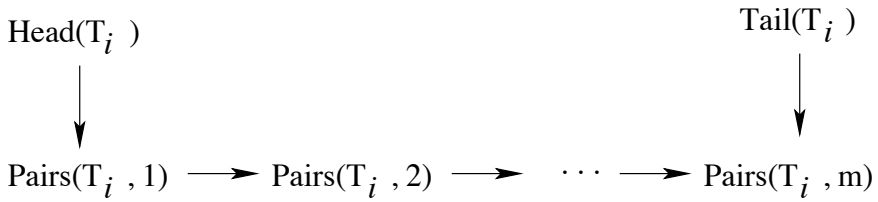


Figure 26: Layout of pairs for thread $T_i$ with $m$ levels.

to the pair structure, the second points to another predecessor structure, which contains a pointer to the actual predecessor of the pair.

The pairs are stored in a series of linked lists, each thread has its own linked list. The head and tail pointers for each linked list are stored in a shared array so each thread can access every other threads' linked lists. The nodes of the linked lists each contain an array of pair structures that the thread has affinity to. This structure can be seen in Figure 26. Each node also contains an array of predecessor pointer structures that correspond to the pairs it has affinity to.

## 5.2 Optimizations

In addition to pLCS being designed to make heavy use of the MuPC cache, several additional optimizations were made.

In order to allow packing of more pairs into a single cache line, rather than use a separate pruned flag, the sign bit of one of the indexes was used to indicate that the pair was pruned. The pointer to the predecessor is also stored outside of the actual pair structure, to help reduce the size of the pair structure.

Since accessing shared data that a thread has affinity to through a pointer to shared is significantly slower than accessing non-shared memory, wherever possible, shared pointers were cast to local pointers. Also whenever an invariant non-local pointer is accessed, a local copy of the pointer is made.

Since the pruning rules that are known to be correct are transitive (i.e. if pair $p_2$ can be pruned due to pair $p_1$ and $p_1$ can be pruned due to pair $p_0$, then $p_2$ can be pruned due to $p_0$), when applying pruning rules a pruned pair is never compared to unpruned pairs to see if the pruned pair allows for pruning.

Also, before pruning, pairs within each level are sorted such that pairs with the smallest difference between $i$ and $j$ values are examined first, as they are more likely to be on the LCS.

### 5.2.1 Ineffective Optimizations

Since pruning is $O(n^2)$ with respect to the number of pairs in a level, several approaches were attempted to reduce the cost of pruning. One approach attempted was to not prune every level. This however this leads to larger numbers of pairs in each level, which greatly increased the pruning time in each level.

Another approach was to prune every level, but each thread only compares its own pairs to one other threads' pairs. Since only two threads are comparing their pairs, the cost is $O((\frac{n}{T})^2)$ instead of $O\left(\frac{n^2}{T}\right)$. This has the advantage of performing far fewer comparisons per level, but is less effective at pruning, so the total number of pairs grows. In limited testing the effectiveness of this change was inconclusive.

# 6 Performance

The UPC implementation of the pLCS algorithm was run on different sizes of randomly generated sequences. An LCS of each pair of sequences was computed using up to 24 nodes of a dual-core, dual-processor Pentium cluster with an Infiniband network using the MuPC runtime system. The sequential run time reported here is for this implementation on one thread. Care was taken to ensure that all references to pointers to shared were cast as local. Correctness was checked against a sequential implementation of the standard dynamic programming algorithm. (It was this checking that led to the two counterexamples in Section 2.3.)

Figure 27 shows that an efficiency of almost 50% (a speedup of 6 on 14 nodes) is achieved for a pair of sequences of size 10,000. Figure 28 shows that this implementation of pLCS is critically dependent on the runtime cache.

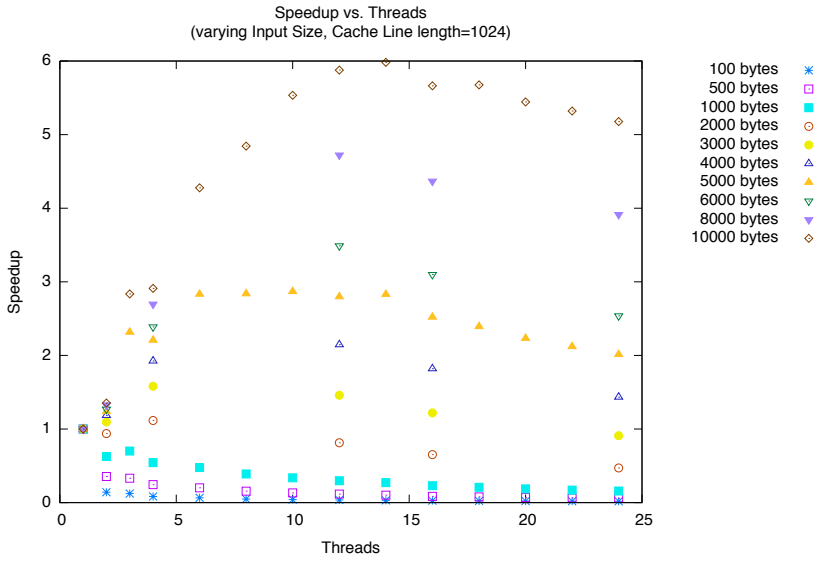


Figure 27: Speedup for various input sizes and threads with cache enabled.

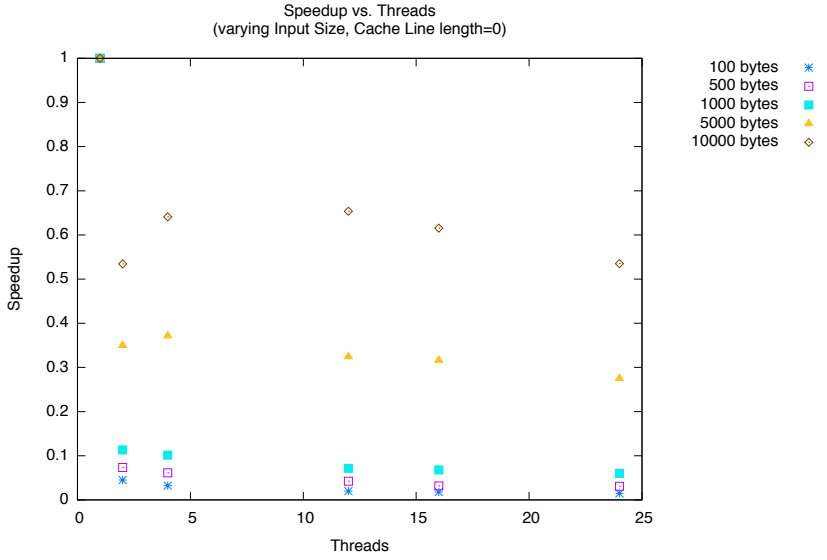


Figure 28: Speedup for various input sizes and threads with cache disabled.

## 6.1 Measurements and Observations

As the input size increases, the runtime increases rather quickly when the cache is disabled, as Figure 29 shows for cache lines of zero bytes. However with the cache enabled the increase is far less significant, as Figure 29 also shows.

When the input size is large, as shown in Figure 30, increasing the number of threads can greatly reduce the total run-time. However, increasing the number of threads only helps to a certain point.

As Figure 29 indicates, enabling the MuPC cache, greatly reduces the run-time for pLCS with large inputs.

With larger input sizes, the cache does help tremendously, as Figure 31 clearly shows. The size of cache lines however does not seems to affect performance as much as might be expected. Cache lines of 1024 bytes seem to be optimal for larger inputs. Increasing to 2048 bytes does not help much, and in some cases even hurts performance. Above 2048 bytes lead to stability problems, and were not studied in great detail.

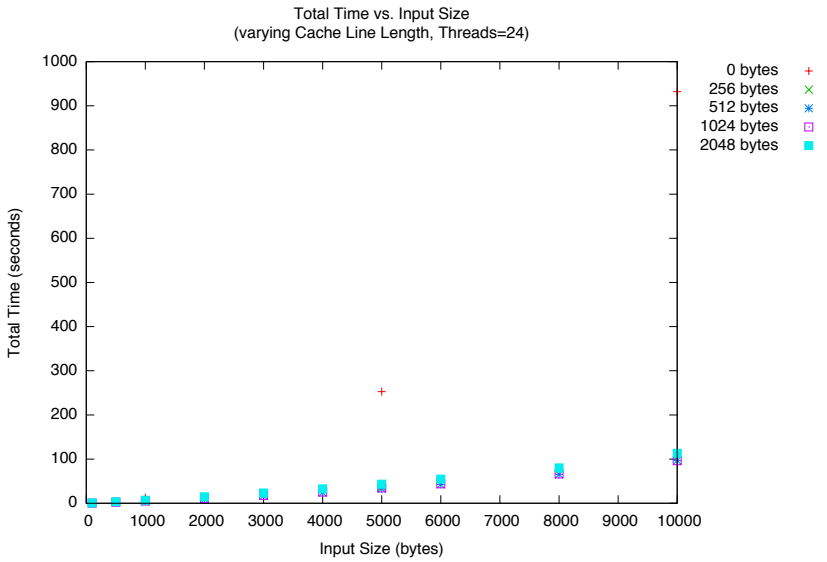


Figure 29: Total execution time versus input size.

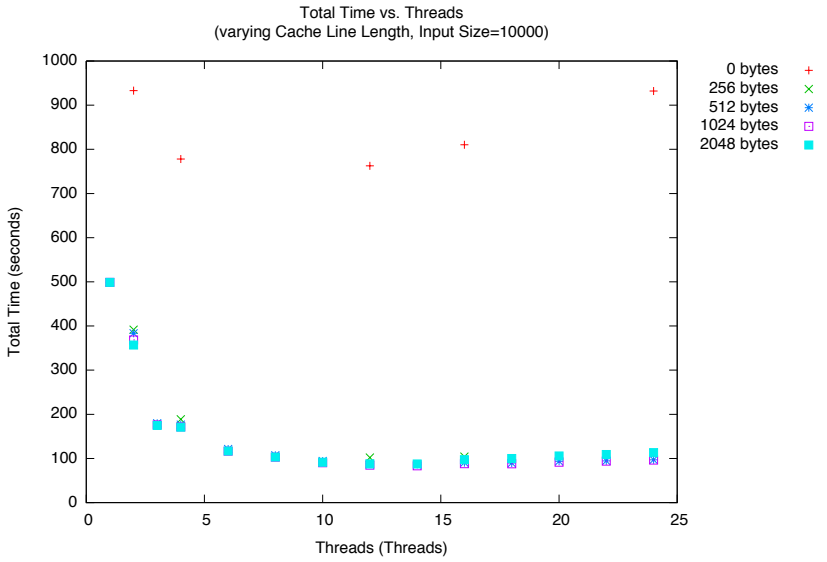


Figure 30: Total time with different numbers of threads for large inputs.

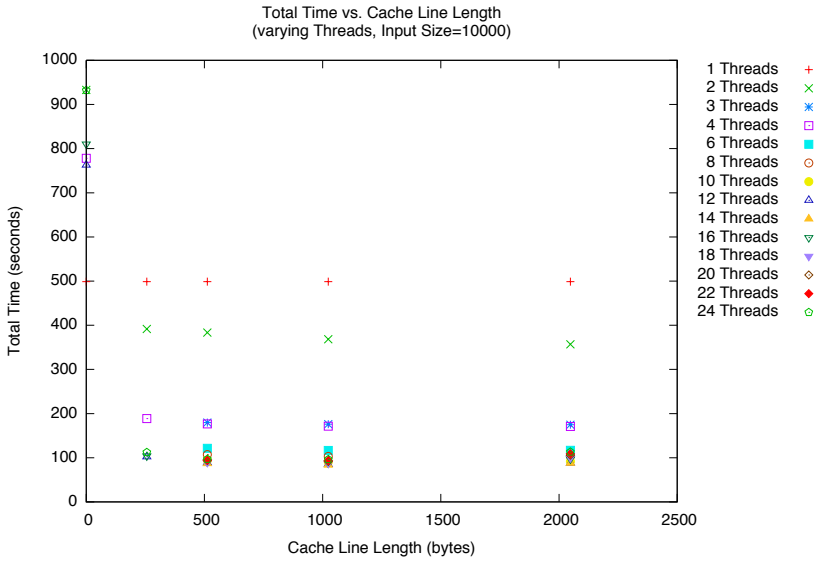


Figure 31: Time versus cache size for large inputs.

Since a major factor in the total cost of pLCS is the cost of pruning, and the cost of pruning is $O(n^2)$ with respect to the number of pairs in a level, it is important to understand how the number of pairs varies through normal execution with pruning. Figure 32 shows how the number of pairs varies with two random input strings. The quadratic shape of the pairs versus level curve appeared in a wide variety of tests on random strings.

Since the quadratic shape seen in Figure 32 seems common for many inputs, the behavior of the peak can give some insight into the behavior of the pLCS algorithm. As Figure 33 shows, the relationship between the input size and the maximum number of pairs that will appear in a level, is linear.

Even with pruning, the number of pairs within a level can grow to be quite large, pLCS was designed to make maximum use of the cache, specifically while pruning. As Figure 34 shows, the cache hit rate is very high for all but very small numbers of pairs. However, as can also be seen in Figure 34, the number of pairs needed to achieve a high hit rate increases with the number of threads. For 2 threads the hit rate goes up the fastest, and for larger numbers of threads more pairs are needed to achieve the same hit rate. For a single thread, the cache is automatically disabled, and thus reports a hit rate of zero.

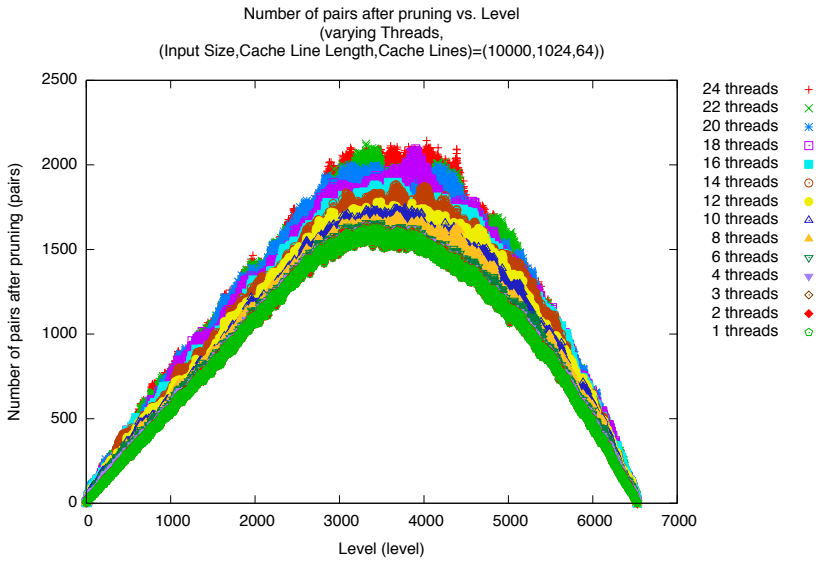


Figure 32: Pairs within a level over the course of algorithms execution.

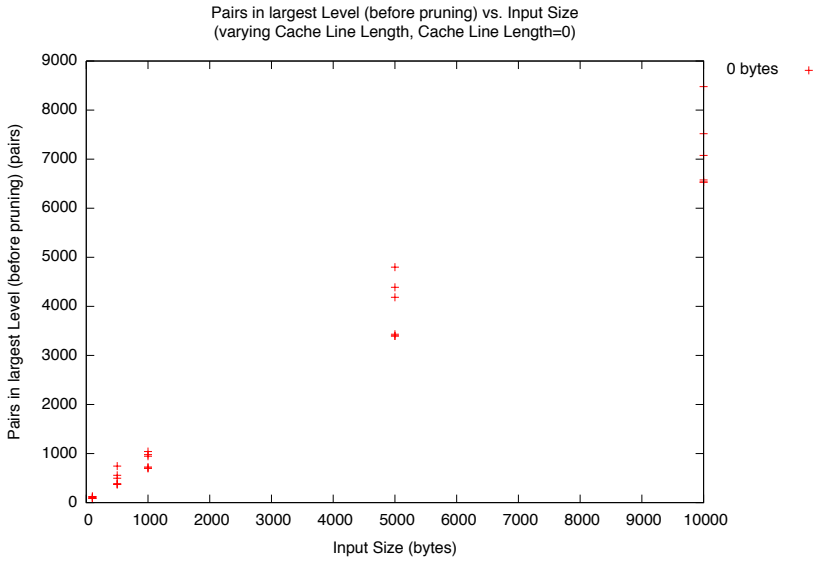


Figure 33: Maximum pairs within a level versus input size.

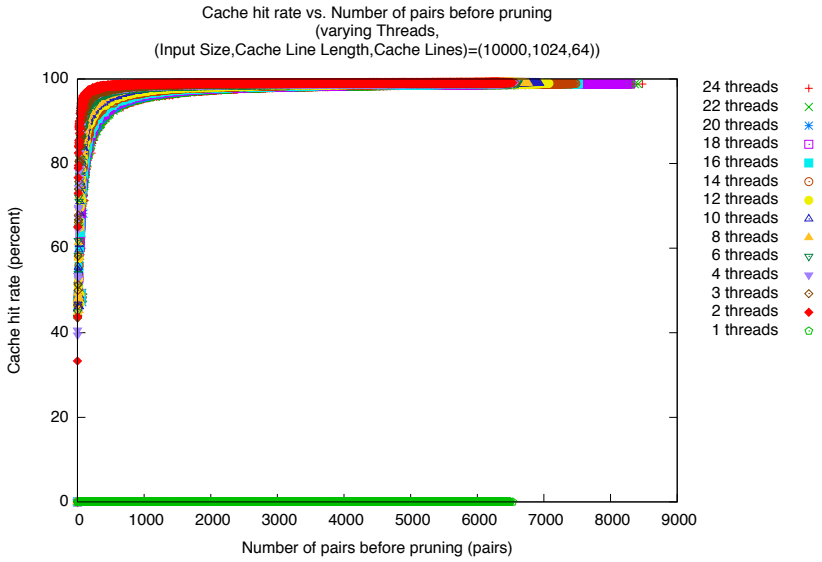


Figure 34: Cache hit rate versus pairs before pruning.

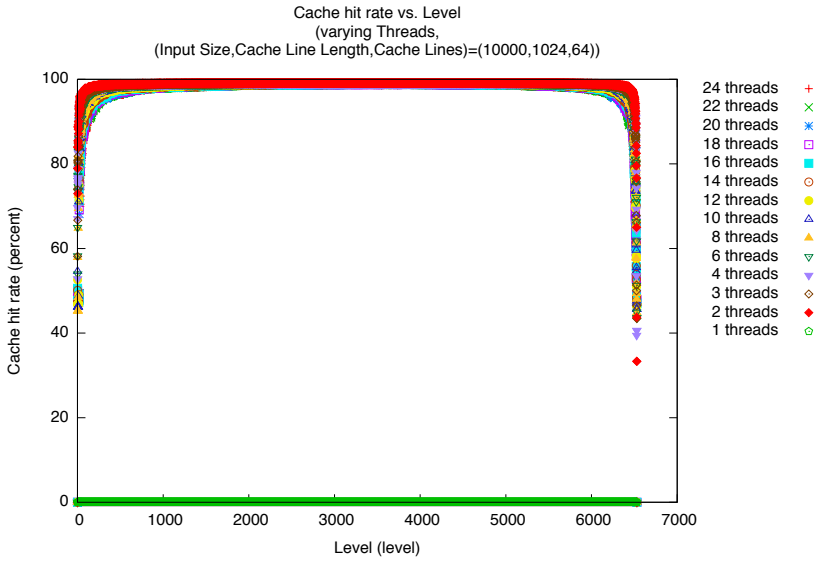


Figure 35: Cache hit rate within each level.

Considering Figures 32 and 34, the hit rate over the course of execution should be very high for most levels. This is exactly what Figure 35 shows.

Since pLCS relies on pruning rules to keep the number of pairs down, threads can end up with differing numbers of pairs at the end of each level. To compensate, pLCS uses a load balancing algorithm, as described in Section 2.5.

While Section 2.5.2 provides a proof that the number of messages needed to rebalance is always less than the number of threads, Figure 36 illustrates this for up to 24 threads on a large input. While Figure 36 only shows the maximum number of messages needed, any number of messages from zero to $T - 1$ are seen, depending on the how unbalanced the data is after each iteration. However the number of messages is less important than the number of communication rounds needed. As Figure 37 shows, while the average number of communication rounds for rebalancing is closely related to number of threads for very small thread counts, for larger threads counts the number of rounds starts to stabilize. Likewise as the input size increases the average number of rounds needed to rebalance grows more slowly for larger inputs, as Figure 38 shows.

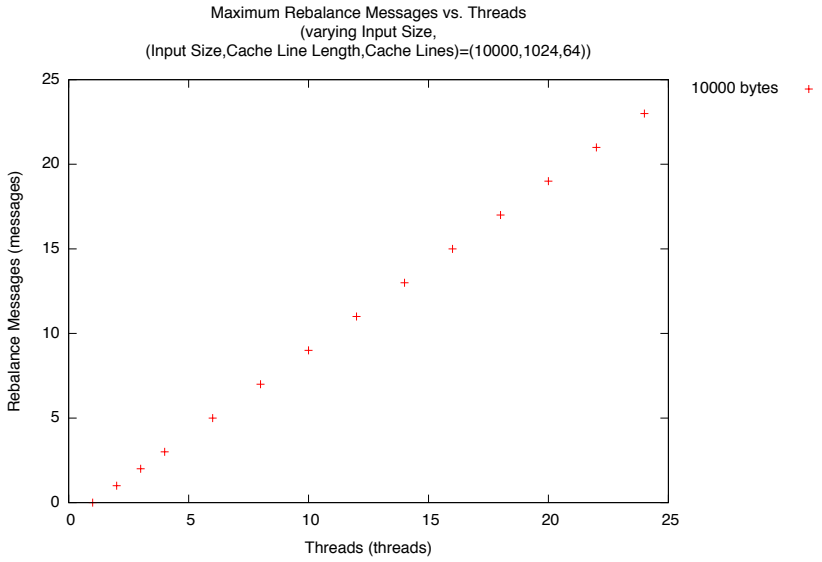


Figure 36: Messages needed versus number of threads.

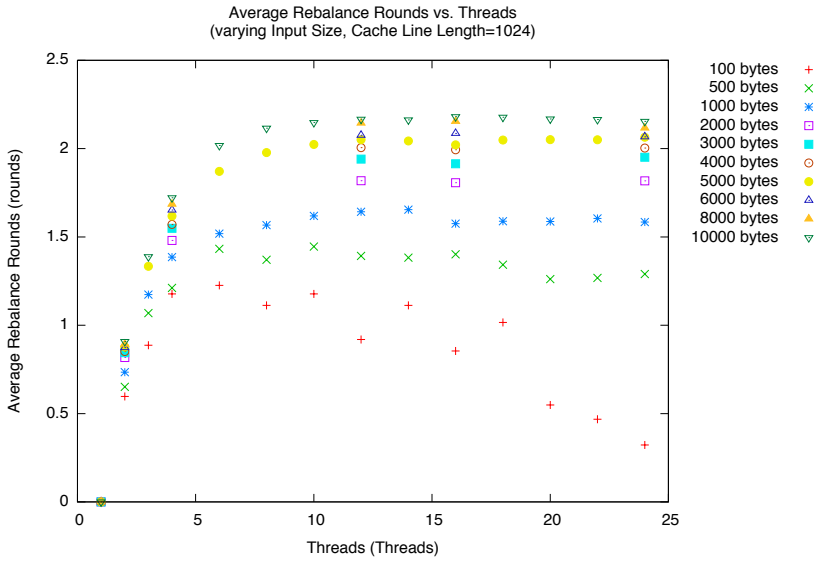


Figure 37: Average rounds needed versus number of threads.

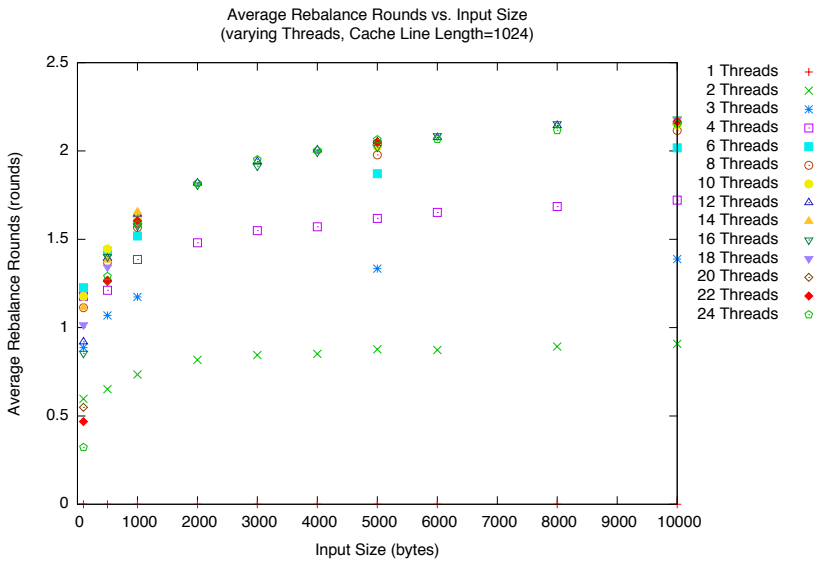


Figure 38: Average rounds needed versus input size.

Since each iteration of pLCS consists of multiple phases, each of which having a different cost function associated with it, the amount of time spent in each phase can vary considerably.

Figures 39 through 44 show the percentage of time spent in each phase for large inputs. The percentage of time needed to create new entries in the linked list of levels is shown in Figure 39. The percentage of time needed to fill the new level with pairs using the successor tables is shown in Figure 40. The percentage of time needed to sort the pairs within each thread for faster pruning is shown in Figure 41. The percentage of time needed to prune the pairs within each level is shown in Figure 42. The percentage of time needed to rebalance the number of pairs in all threads is shown in Figure 43. The percentage of time needed to check if the a level is empty, and LCS has been found, in each level is shown in Figure 44.

For large inputs, the phase that dominates execution is clearly pruning, as Figure 42 demonstrates. As the number of threads increases though, the percentage of time spent pruning decreases and the percentage of time spent rebalancing increases, as Figures 42 and 43 show. In Figure 42, the largest percentages are for 1 thread and the smallest percentages are for 24 threads, the rest of the points fall in the middle. A similar trend can be seen in Figures 40 and 41, however mostly at the start and end of execution where the number of pairs, and thus pruning time is considerably less. In Figure 43 the opposite holds, here the lowest percentages are for a single thread and increasing the threads increases the percentage of time spent rebalancing with 24 threads taking the largest percentage of time to rebalance. This trend can also be seen to a lesser extent in Figures 39 and 44.

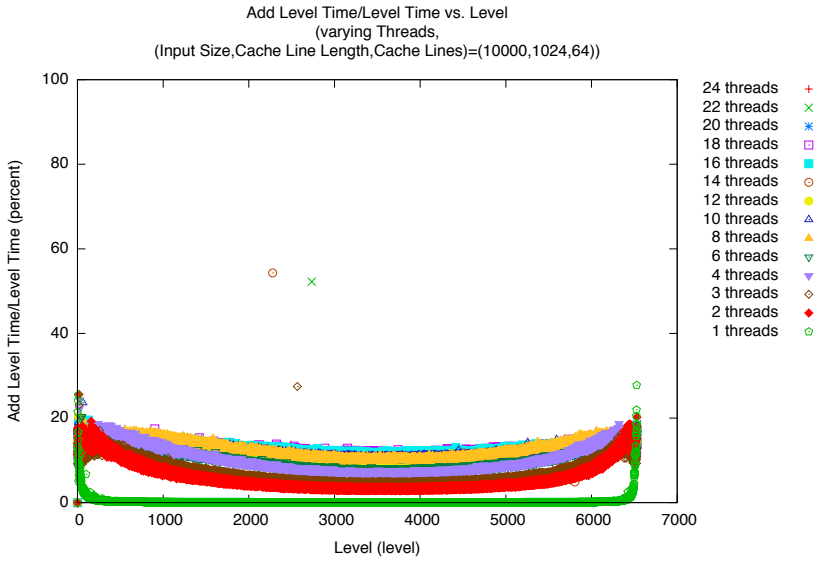


Figure 39: Percent of time needed to add a level for each level for large input size.

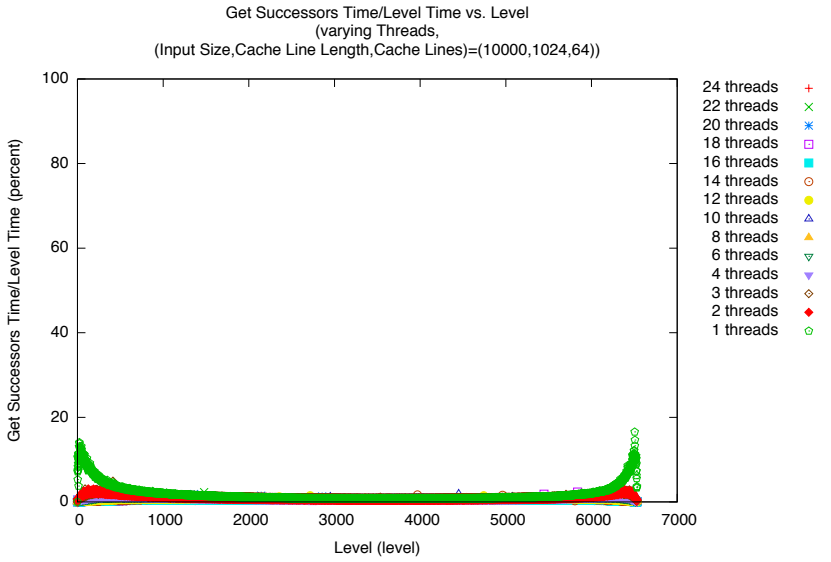


Figure 40: Percent of time needed to find successors for each level for large input size.

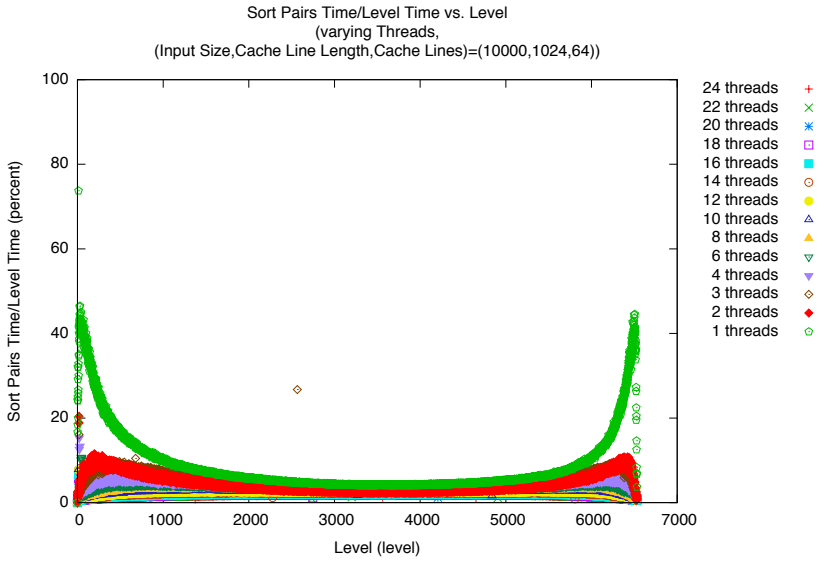


Figure 41: Percent of time needed to sort pairs within each level for large input size.

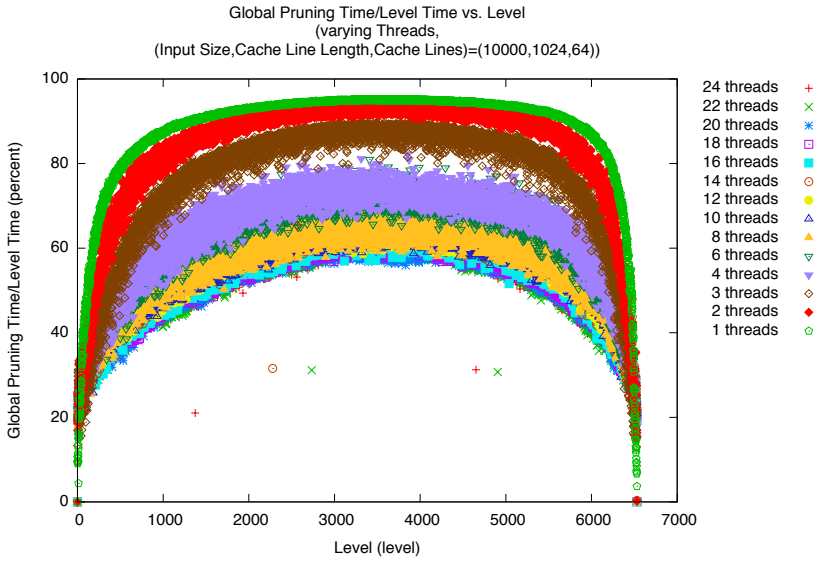


Figure 42: Percent of time needed to prune each level for large input size.

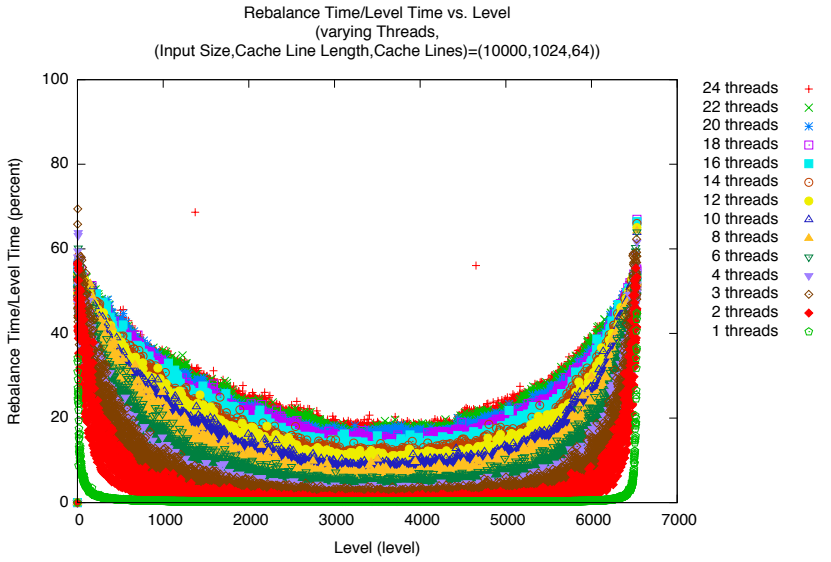


Figure 43: Percent of time needed to rebalance each level for large input size.

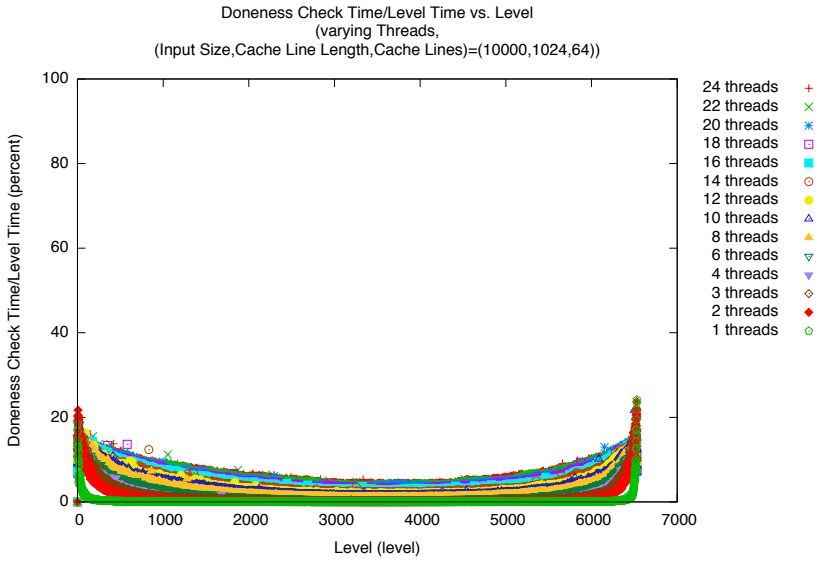


Figure 44: Percent of time needed to check for completion at each level for large input size.

# 7 Conclusions and Future Work

This work examined a parallel algorithm for the longest common subsequence problem based on an algorithm proposed by Liu *et al.*. The major contributions of this work are the development of a correct version of that algorithm, a PGAS implementation of that algorithm that exhibits speedup, and an implementation that does not use any shared memory bulk copy operations. The UPC implementation studied here made good use of a runtime cache because the pruning rules of the algorithm heavily reuse remote data and only writes back data after it is done with it. In particular, performance was significantly degraded when the cache was turned off. The programmer of an MPI version of such a code would have to implement a runtime cache explicitly in order to achieve the same reuse benefits.

Future work should consider the scalability of pLCS on platforms larger than 24 nodes and on a wider range of test data, such as real genome sequences rather than randomly generated sequences. Performance should be measured using other runtime systems to determine whether optimizations in those systems are relevant to this implementation.

It is possible that more aggressive pruning rules can be found. (A generalization of Claims 1 and 2 was implemented and found to provide much improved performance but at the price of correctness.) We speculate that there are additional pruning rules that will improve performance.

Additional future work could place bounds on the error introduced by pruning rules that do not lead to exact results.

# References


[1] A. Aggarwal and J. Park. Notes on searching in multidimensional monotone arrays. *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 497–512, Oct 1988.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, October 1990.

[3] Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.

[4] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. *String Processing and Information Retrieval, International Symposium on*, 0:39, 2000.

[5] Jik H. Chang, Oscar H. Ibarra, and Michael A. Palis. Parallel parsing on a one-way array of finite-state machines. *Computers, IEEE Transactions on*, C-36(1):64–75, Jan. 1987.

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

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

[8] Elizabeth E. Edmiston, Nolan G. Core, Joel H. Saltz, and Roger M. Smith. Parallel processing of biological sequence comparison algorithms. *Int. J. Parallel Program.*, 17(3):259–275, 1988.

[9] Valerio Freschi and Alessandro Bogliolo. Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism. *Inf. Process. Lett.*, 90(4):167–173, 2004.

[10] Adam R. Galper and Douglas L. Brutlag. Parallel similarity search and alignment with the dynamic programming method. Technical report, Stanford University, California, 1990.

[11] J. Y. Guo and F. K. Hwang. An almost-linear time and linear space algorithm for the longest common subsequence problem. *Inf. Process. Lett.*, 94(3):131–135, 2005.

[12] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[13] Michael Höhl, Stefan Kurtz, and Enno Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18(suppl_1):S312–320, 2002.

[14] Wei Liu, Yixin Chen, Ling Chen, and Ling Qin. A fast parallel longest common subsequence algorithm based on pruning rules. *Computer and Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums on*, 1:27–34, June 2006.

[15] Mi Lu and Hua Lin. Parallel algorithms for the longest common subsequence problem. *Parallel and Distributed Systems, IEEE Transactions on*, 5(8):835–848, Aug 1994.

[16] Guillaume Luce and Jean Frédáéric Myoupo. Systolic-based parallel architecture for the longest common subsequences problem. *Integr. VLSI J.*, 25(1):53–70, 1998.

[17] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, 1978.

[18] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[19] Gene Myers. A four russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):432–448, 1992.

[20] Jean-Frédéric Myoupo and David Semé. Time-efficient parallel algorithms for the longest common subsequence and related problems. *J. Parallel Distrib. Comput.*, 57(2):212–223, 1999.

[21] K. Nandan Babu and S. Saxena. Parallel algorithms for the longest common subsequence problem. *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pages 120–125, Dec 1997.

[22] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[23] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988.

[24] Y. Robert and M. Tchuente. A systolic array for the longest common subsequence problem. *Information Processing Letters*, (21):191–198, 1985.

[25] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[26] J. D. Ullman, A. V. Aho, and D. S. Hirschberg. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, 23(1):1–12, 1976.

[27] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[28] Fa Zhang, Xiang-Zhen Qiao, and Zhi-Yong Liu. A parallel smith-waterman algorithm based on divide and conquer. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 162–169, 2002.

[29] Z. Zhang, J. Savant, and S. Seidel. A UPC Runtime System based on MPI and POSIX Threads. In *Proc. of 14th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2006)*, 2006.