Parallelizing Dynamic Programming Through Rank Convergence

Saeed Maleki
Univerity of Illinois at Urbana-Champaign
maleki1@illinois.edu

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

Abstract

This paper proposes an efficient parallel algorithm for an important class of dynamic programming problems that includes Viterbi, Needleman-Wunsch, Smith-Waterman, and Longest Common Subsequence. In dynamic programming, the subproblems that do not depend on each other, and thus can be computed in parallel, form stages or *wavefronts*. The algorithm presented in this paper provides additional parallelism allowing multiple stages to be computed in parallel despite dependences among them. The correctness and the performance of the algorithm relies on rank convergence properties of matrix multiplication in the tropical semiring, formed with plus as the multiplicative operation and max as the additive operation.

This paper demonstrates the efficiency of the parallel algorithm by showing significant speed ups on a variety of important dynamic programming problems. In particular, the parallel Viterbi decoder is up-to 24× faster (with 64 processors) than a highly optimized commercial baseline.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; F.1.2 [*Modes of Computation*]: Parallelism and concurrency

Keywords Parallelism; Dynamic Programming; Tropical Semiring; Wavefront; Viterbi; Smith-Waterman;

1. Introduction

Dynamic programming [4] is a method to solve a variety of important optimization problems in computer science, economics, genomics, and finance. Figure 1 describes two such examples: Viterbi, which finds the most-likely path through a hidden-Markov model for a sequence of observations and LCS, which finds the longest common subsequence between two input strings. Dynamic programming algorithms proceed by recursively solving a series of subproblems, usually represented as cells in a table as shown in the figure. The solution to a subproblem is constructed from solutions to an appropriate set of subproblems, as shown by the respective recurrence relation in the figure.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.
Copyright © 2014 ACM 978-1-4503-2656-8/14/02…$15.00.
http://dx.doi.org/10.1145/2555243.2555264

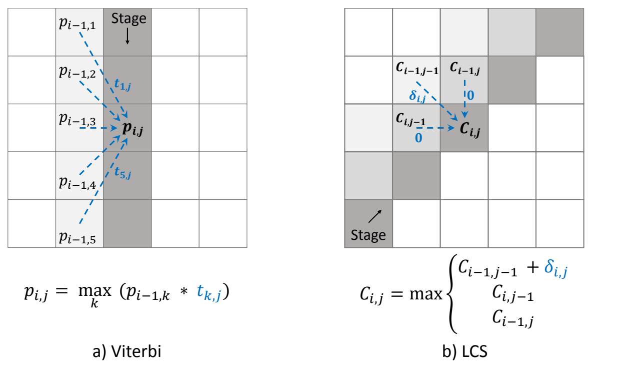


Figure 1: Dynamic programming examples with dependences between stages.

These data-dependences naturally group subproblems into *stages* whose solutions do not depend on each other. For example, all subproblems in a column form a stage in Viterbi and all subproblems in an anti-diagonal form a stage in LCS. A predominant method for parallelizing dynamic programming is *wavefront* parallelization [20], which computes all subproblems within a stage in parallel.[1]

In contrast, this paper breaks data-dependences *across* stages and fixes up incorrect values later in the algorithm. Therefore, this approach exposes parallelism for a class of dynamic programming algorithms we call *linear-tropical dynamic programming* (LTDP). A LTDP computation can be viewed as performing a sequence of matrix multiplications in the tropical semiring where the semiring is formed with $+$ as the multiplicative operator and $\max$ as the additive operator. This paper demonstrates that several important optimization problems such as Viterbi, LCS, Smith-Waterman,

[1] The definition of wavefront parallelism used here is more general and includes the common usage where a wavefront performs computations across logical iterations as in the LCS example in Figure 1(a).

and Needleman-Wunsch (the latter two are used in bioinformatics for sequence alignment) belong to LTDP. To efficiently break data-dependences across stages, the algorithm uses *rank convergence*, a property by which the rank of a sequence of matrix products in the tropical semiring is likely to converge to 1.

A key advantage of our parallel algorithm is its ability to simultaneously use both the coarse-grained parallelism across stages and the fine-grained wavefront parallelism within a stage. Moreover, the algorithm can reuse existing highly-optimized implementations that exploit wavefront parallelism with little modification. As a consequence, our implementation achieves multiplicative speed ups over existing implementations. For instance, the parallel Viterbi decoder is up-to $24\times$ faster with 64 cores than a state-of-the-art commercial baseline [25]. This paper demonstrates similar speed ups for other LTDP instances studied in this paper.

2. Background

In linear algebra, a matrix-vector multiplication maps a vector from an input space to an output space. If the matrix is of low rank, the matrix maps the vector to a subspace of the output space. In particular, if the matrix has rank 1, then it maps all input vectors to a *line* in the output space. These geometric intuitions hold even when one changes the meaning of the sum and multiplication operators (say to $\max$ and $+$, respectively), as long as the new meaning satisfies the following rules.

Semirings A semiring is a five-tuple $(D, \oplus, \otimes, \mathbb{0}, \mathbb{1})$, where D is the domain of the semiring that is closed under the additive operation $\oplus$ and the multiplicative operation $\otimes$. The two operations satisfy the following properties:

- $(D, \oplus, \mathbb{0})$ forms a commutative monoid with $\mathbb{0}$ as the identity
  - associativity: $\forall x, y, z \in D : (x \oplus y) \oplus z = x \oplus (y \oplus z)$
  - identity: $\forall x \in D : x \oplus \mathbb{0} = x$
  - commutativity: $\forall x, y \in D : x \oplus y = y \oplus x$
- $(D, \otimes, \mathbb{1})$ forms a monoid with $\mathbb{1}$ as the identity
  - associativity: $\forall x, y, z \in D : (x \otimes y) \otimes z = x \otimes (y \otimes z)$
  - identity: $\forall x \in D : x \otimes \mathbb{1} = \mathbb{1} \otimes x = x$
- $\otimes$ left- and right-distributes over $\oplus$
  - $\forall x, y, z \in D : x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$
  - $\forall x, y, z \in D : (y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$
- $\mathbb{0}$ is an annihilator for $\otimes$
  - $\forall x \in D : x \otimes \mathbb{0} = \mathbb{0} \otimes x = \mathbb{0}$

Tropical Semiring The semiring $(\mathbb{R} \cup \{-\infty\}, \max, +, -\infty, 0)$ with the real numbers extended with $-\infty$ as the domain, $\max$ as the additive operation $\oplus$, and $+$ as the multiplicative operation $\otimes$ is called the tropical semiring. All properties of a semiring hold with $-\infty$ as the additive identity $\mathbb{0}$ and 0 as the multiplicative identity $\mathbb{1}$. Alternately, one can reverse the sign of every element in the domain and change the additive operation to $\min$.

Matrix Multiplication Let $A_{n \times m}$ denote a matrix with n rows and m columns with elements from the domain of the tropical semiring. Let $A[i, j]$ denote the element of A at the ith row and jth column. The matrix product of $A_{l \times m}$ and $B_{m \times n}$ is $A \odot B$, a $l \times n$ matrix defined such that

$$\begin{aligned}(A \odot B)[i, j] &= \bigoplus_{1 \le k \le m} (A[i, k] \otimes B[k, j]) \\ &= \max_{1 \le k \le m} (A[i, k] + B[k, j])\end{aligned}$$

Note, this is the standard matrix product with multiplication replaced by $+$ and addition replaced by $\max$.

The transpose of $A_{n \times m}$ is the matrix $A^{\intercal}_{m \times n}$ such that $\forall i, j : A^{\intercal}[i, j] = A[j, i]$. Using standard terminology, we will denote a $v_{n \times 1}$ matrix as the column vector $\vec{v}$, a $v_{1 \times n}$ matrix as the row vector $\vec{v}^{\intercal}$, and $x_{1 \times 1}$ matrix simply as the scalar x. This terminology allows us to extend the definition of matrix-matrix multiplication above to matrix-vector, scalar-matrix, and scalar-vector multiplication appropriately. Also, $\vec{v}[i]$ is the ith element of a vector $\vec{v}$. The following lemma follows from the associativity, distributivity, and commutativity properties of $\otimes$ and $\oplus$ in a semiring.

Lemma 1. *Matrix multiplication is associative in semirings*

$$(A \odot B) \odot C = A \odot (B \odot C)$$

Parallel Vectors Two vectors $\vec{u}$ and $\vec{v}$ are parallel in the tropical semiring, denoted as $\vec{u} \parallel \vec{v}$, if there exist scalars x and y such that $\vec{u} \odot x = \vec{u} \odot y$. Intuitively, parallel vectors in tropical semiring $\vec{u}$ and $\vec{v}$ differ by a constant offset. For instance, $[1\,0\,2]^{\intercal}$ and $[3\,2\,4]^{\intercal}$ are parallel vectors differing by an offset 2. Note that the definition above requires two scalars as $-\infty$ does not have a multiplicative inverse in the tropical semiring.

Matrix Rank The rank of a matrix $M_{m \times n}$, denoted by $\mathrm{rank}(M)$, is the smallest number r such that there exist matrices $C_{m \times r}$ and $R_{r \times n}$ whose product is M. In particular, a rank-1 matrix is a product of a column vector and a row vector. There are alternate ways to define the rank of a matrix in semirings, such as the number of linearly independent rows or columns in a matrix. While such definitions coincide in fields (which have inverses for $\oplus$ and $\otimes$), they are not equivalent in semirings [7].

Lemma 2. *For any vectors $\vec{u}$ and $\vec{v}$ and a matrix A of rank 1, $A \odot \vec{u} \parallel A \odot \vec{v}$*

Intuitively, this lemma states that a rank-1 matrix maps all vectors to a line. If $\mathrm{rank}(A) = 1$ then it is a product of some column vector $\vec{c}$ and a row vector $\vec{r}^\intercal$. For any vectors $\vec{u}$ and $\vec{v}$:

$$A \odot \vec{u} = (\vec{c} \odot \vec{r}^\intercal) \odot \vec{u} = \vec{c} \odot (\vec{r}^\intercal \odot \vec{u}) = \vec{c} \odot x_u$$
$$A \odot \vec{v} = (\vec{c} \odot \vec{r}^\intercal) \odot \vec{v} = \vec{c} \odot (\vec{r}^\intercal \odot \vec{v}) = \vec{c} \odot x_v$$

for appropriate scalars x_u and x_v. As an example, consider

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} \qquad \vec{u} = \begin{bmatrix} 1 \\ -\infty \\ 3 \end{bmatrix} \qquad \vec{v} = \begin{bmatrix} -\infty \\ 2 \\ 0 \end{bmatrix}$$

$A = [1\,2\,3]^\intercal \odot [0\,1\,2]$ is rank-1. $A \odot \vec{u} = [6\,7\,8]^\intercal$ and $A \odot \vec{v} = [4\,5\,6]^\intercal$ which are parallel with a constant offset 2. Also note that all rows in a rank-1 matrix are parallel to each other.

3. Linear-Tropical Dynamic Programming

Dynamic programming is a method for solving problems that have optimal substructure — the solution to a problem can be obtained from the solutions to a set of its overlapping subproblems. This dependence between subproblems is captured by a recurrence equation. Classic dynamic programming implementations solve the subproblems iteratively applying the recurrence equation in an order that respects the dependence between subproblems.

LTDP Definition A dynamic programming problem is linear-tropical dynamic programming (LTDP), if (a) the subproblems can be grouped into a sequence of stages such that the solution to a subproblem in a stage only depends on the solutions in the previous stage and (b) this dependence is linear in the tropical semiring. In other words, $s_i[j]$, the solution to subproblem j in stage i of LTDP, is given by the recurrence equation

$$s_i[j] = \max_k (s_{i-1}[k] + A_i[j,k]) \tag{1}$$

for appropriate constants $A_i[j,k]$. This linear dependence allows us to view LTDP as computing a sequence of vectors $\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n$, where

$$\vec{s}_i = A_i \odot \vec{s}_{i-1} \tag{2}$$

for an appropriate matrix of constants A_i derived from the recurrence equation. In this equation, we will call $\vec{s}_i$ as the *solution vector* at stage i and call A_i as the *transformation matrix* at stage i. Also, $\vec{s}_0$ is the initial solution vector obtained from the base case of the recurrence equation.

Predecessor Product Once all of the subproblems are solved, finding the solution to the underlying optimization problem of LTDP usually involves tracing the *predecessors* of subproblems. A predecessor of a subproblem is the subproblem for which the maximum in Equation 1 is reached. For ease of exposition, we define the *predecessor product* of a matrix A and a vector $\vec{s}$ as the vector $A \star \vec{s}$ such that

$$(A \star \vec{s})[j] = \arg\max_k (\vec{s}[k] + A[j,k])$$

```
1  LTDP_Seq (vector s₀, matrix A₁..Aₙ) {
2    vector pred[1..n];  vector res;
3    // forward
4    s = s₀;
5    for i in (1..n) {
6      pred[i] = Aᵢ ⋆ s; // pred[i] = $\vec{p}_i$
7      s       = Aᵢ ⊙ s; // s = $\vec{s}_i$
8    }
9    // backward
10   res[n+1] = 0;  // res = $\vec{r}$
11   for i in (n..1) {
12     res[i] = pred[i][res[i+1]];        }
13   return res;                          }
```

Figure 2: LTDP implementation that computes the stages sequentially. An implementation can possibly employ wavefront parallelization within a stage.

Note the similarity between this definition and Equation 1. We assume that ties in $\arg\max$ are broken deterministically. The following lemma shows that predecessor products do not distinguish between parallel vectors, a property that will be useful later.

Lemma 3. $\vec{u} \parallel \vec{v} \implies \forall A : A \star \vec{u} = A \star \vec{v}$

This follows from the fact that parallel vectors in the tropical semiring differ by a constant and that $\arg\max$ is invariant when a constant is added to all its arguments.

Sequential LTDP Figure 2 shows the *sequential* algorithm for LTDP phrased in terms of matrix multiplications and predecessor products. This algorithm is deemed sequential because it computes the stages one after the other based on the data-dependence in Equation 1. However, the algorithm can utilize wavefront parallelism to compute the solutions within a stage in parallel.

The inputs to the sequential algorithm are the initial solution vector $\vec{s}_0$ and transformation matrices $A_1, \ldots, A_n$, which respectively capture the base and inductive case of the LTDP recurrence equation. The algorithm consists of a forward phase and a backward phase. The forward phase computes the solutions in each stage $\vec{s}_i$ iteratively. In addition, it computes the predecessor product $\vec{p}_i$ that determines the predecessor for each solution in a stage. The backward phase iteratively follows the predecessors computed in the forward phase. The algorithm assumes that the first subproblem in the last stage, $\vec{v}_n[0]$, contains the desired solution to the underlying optimization problem. Accordingly, the backward phase starts with 0 in Line 10. The resulting vector `res` is the solution to the optimization problem at hand (e.g., the longest-common-subsequence of the two input strings).

The exposition above consciously hides a lot of details in the $\odot$ and $\star$ operators. An implementation does not need to represent the solutions in a stage as a vector and perform matrix-vector operations. It might statically know that the current solution depends on some of the subproblems in the previous stage (a sparse matrix) and only accesses those.

Finally, as mentioned above, an implementation might use wavefront parallelism to compute the solutions in a stage in parallel. All these implementation details are orthogonal to how the parallel algorithm described in this paper parallelizes across stages.

4. Parallel LTDP Algorithm

This section describes an efficient algorithm for parallelizing the sequential algorithm in Figure 2 across stages.

4.1 Breaking Data-Dependences Across Stages

Viewing LTDP computation as matrix multiplication in the tropical semiring provides a way to break data-dependences among stages. Consider the solution vector at the last stage $\vec{s}_n$. From Equation 2, we have

$$\vec{s}_n = A_n \odot A_{n-1} \ldots A_2 \odot A_1 \odot \vec{s}_0$$

Standard techniques [11, 16] can parallelize this computation using the associativity of matrix multiplication. For instance, two processors can compute the partial products $A_n \odot \ldots \odot A_{n/2+1}$ and $A_{n/2} \odot \ldots \odot A_1$ in parallel, and multiply their results with $\vec{s}_0$ to obtain $\vec{s}_n$.

However, doing so converts a sequential computation that performs matrix-vector multiplications to a parallel computation that performs matrix-matrix multiplications. This results in a parallelization overhead linear in the size of the stages and thus requires linear number of processors to observe constant speed ups. In practice, the size of stages can easily be hundreds or larger and thus is not practical on real problems and hardware.

The key contribution of this paper is a parallel algorithm that avoids the overhead of matrix-matrix multiplications. This algorithms relies on the convergence of matrix rank in the tropical semiring as discussed below. Its exposition requires the following definition.

Partial Product For a given LTDP instance, the partial product $M_{i\to j}$, defined for stages $j \geq i$, is given by

$$M_{i\to j} = A_j \odot \ldots A_{i+1} \odot A_i$$

Partial product determines how a later stage j depends on stage i as $\vec{s}_j = M_{i\to j} \odot \vec{s}_i$.

4.2 Rank Convergence

Rank of the product of two matrices is not greater than the rank of the individual matrices.

$$\text{rank}(A \odot B) \leq \min(\text{rank}(A), \text{rank}(B)) \qquad (3)$$

This is because, if $\text{rank}(A) = r$, then $A = C \odot R$ for some matrix C with r columns. Thus, $A \odot B = (C \odot R) \odot B = C \odot (R \odot B)$ implying that $\text{rank}(A \odot B) \leq \text{rank}(A)$. Similar argument shows that $\text{rank}(A \odot B) \leq \text{rank}(B)$.

Equation 3 implies that for stages $k \geq j \geq i$

$$\text{rank}(M_{i\to k}) \leq \text{rank}(M_{i\to j}) \leq \text{rank}(A_i)$$

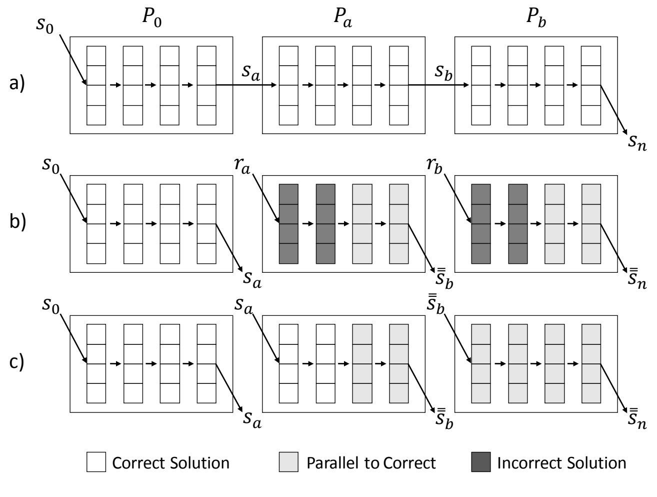


Figure 3: Parallelization algorithm using rank convergence.

In effect, as the LTDP computation proceeds, the rank of the partial products will never increase. Theoretically, there is a possibility that the ranks do not decrease. However, we have only observed this for carefully crafted problem instances that are unlikely to occur in practice. On the contrary, the rank of these partial products is likely to converge to 1, as will be demonstrated in Section 6.1.

Consider a partial product $M_{i\to j}$ whose rank is 1. Intuitively, this implies a *weak* dependence between stages i and j. Instead of the actual solution vector, $\vec{s}_i$, say the LTDP computation starts with a different vector $\vec{t}_i$ at stage i. From Lemma 2, the new solution vector at stage j, $\vec{t}_j = M_{i\to j} \odot \vec{t}_i$, is parallel to the actual solution vector $\vec{s}_j = M_{i\to j} \odot \vec{s}_i$. Essentially, the direction of the solution vector at stage j is independent of stage i. The latter stage only determines its magnitude. In the tropical semiring, where the multiplicative operator is $+$, this means that the solution vector at stage j will be off by a constant if one starts stage i with an arbitrary vector.

4.3 Parallel Algorithm Overview

The parallel algorithm uses this insight to break dependences between stages as shown pictorially in Figure 3. The figure uses three processors as an example. Figure 3(a) represents the forward phase of the sequential algorithm described in Figure 2. Each stage is represented as a vertical column of cells and an arrow between stages represents a multiplication with an appropriate transformation matrix. Processor P_0 starts from the initial solution vector s_0 and computes all its stages. Processor P_a waits for s_a, the solution vector in the final stage of P_0, in order to start its computation. Similarly, processor P_b waits for s_b the solution vector at the final stage of P_a.

In the parallel algorithm shown in Figure 3(b), processors P_a and P_b start from *arbitrary* solutions r_a and r_b respectively in parallel with P_0. Of course, the solutions for

```
1   LTDP_Par(vector s_0, matrix A_1..A_n) {
2     vector s[1..n];  vector pred[1..n];
3     vector conv;
4     // proc p owns stages (l_p..r_p]
5     ∀p: l_p = n/P*(p-1); r_p = n/P*p;
6     // parallel forward phase
7     parallel.for p in (1..P) {
8       local s = (p == 1 ? s_0 : nz);
9       for i in (l_p+1..r_p) {
10        pred[i]  = A_i ⋆ s;
11        s = s[i] = A_i ⊙ s;     }}
12    ----- barrier -----
13    do {  // till convergence (fix up loop)
14      parallel.for p in (2..P) {
15        conv[p] = false;
16        // obtain final soln from prev proc
17        s = s[l_p];
18        for i in (l_p+1..r_p) {
19          pred[i] = A_i ⋆ s;
20          s       = A_i ⊙ s;
21          if( s is parallel to s[i] ) {
22            conv[p] = true;
23            break;                    }
24          s[i] = s;                   }}
25      ----- barrier -----
26      conv = ⋀_p conv[p];
27    } while (!conv);
28
29    //parallel backward phase is in Figure 5
30    return Backward_Par(pred);
```

Figure 4: Parallel algorithm for the forward Pass of LTDP that relies on rank convergence for efficiency. All interprocessor communication is shown in magenta.

the stages computed by P_a and P_b will start out as completely wrong (shaded dark in the figure). However, if rank convergence occurs then these erroneous solution vectors will eventually become parallel to the actual solution vectors (shaded gray in the figure). Thus, P_a will generate some solution vector $\bar{\bar{s}}_b$ parallel to s_b and P_b will generate some solution vector $\bar{\bar{s}}_n$ parallel to s_n.

In a subsequent *fix up phase*, shown in Figure 3(c), P_a uses s_a computed by P_0 and P_b uses $\bar{\bar{s}}_b$ computed by P_1 to fix stages that are not parallel to the actual solution vector at that stage. After the fix up, the solution vectors at each stage are either the same as or parallel to the actual solution vector at those respective stages.

For LTDP, it is not necessary to compute the actual solution vectors. As parallel vectors generate the same predecessor products (Lemma 3), following the predecessors in Figure 3(c) will generate the same solution as the following the predecessors in Figure 3(a).

The next sections describe the parallel algorithm in more detail.

4.4 Parallel Forward Phase

The goal of the parallel forward phase in Figure 4 is to compute a solution vector `s[i]` at stage i that is *parallel* to the actual solution vector $\vec{s}_i$, as shown in Figure 3. During the execution of the algorithm, we say that a stage i has *converged* if `s[i]` computed by the algorithm is parallel to its actual solution vector $\vec{s}_i$.

The parallel algorithm splits the stages equally among P processors such that a processor p owns stages between $\mathtt{l_p}$ (exclusive) and $\mathtt{r_p}$ (inclusive), as shown in line 5. While processor 1 starts its computation from $\vec{s}_0$, other processors start from some vector `nz` (line 8). This initial vector can be arbitrary except none of its entries can be $\mathbb{0} = -\infty$. Section 4.5 explains the importance of this constraint.

The loop starting in line 9 is similar to the sequential forward phase (Figure 2) except that the parallel version additionally stores the computed `s[i]` needed in the convergence loop below.

Consider a processor $p \neq 1$ that owns stages $(\mathtt{l_p} = l \ldots r = \mathtt{r_p}]$. If there exists a stage k in $(l \ldots r]$ such that $\mathrm{rank}(M_{l \to k})$ is 1, then stage k converges, irrespective of the initial vector `nz` (Lemma 2). Moreover, by Equation 3, $\mathrm{rank}(M_{l \to j})$ is 1 for all stages j in $[k \ldots r]$, implying that these stages converge as well (Figure 3(b)). However, processor p is not cognizant of the actual solution vectors and, thus, does not know the value of k or whether such a k exists.

The fix up loop starting at line 13 (fix up phase in Figure 3(c)) fixes stages $i < k$. In this loop, processor p receives the vector at stage l computed by the previous processor $p-1$. (Figure 4 shows all such interprocessor communication in magenta.) Processor p then updates `s[i]` for all stages till the new value becomes parallel to the old value of `s[i]` (line 21). This ensures that all stages owned by p have converged, under the assumption that stage l has converged.

In addition, the Boolean variable `conv[p]` indicates whether processor p advertised a converged value for its last stage to processor $p+1$ at the beginning of the iteration. Thus, when `conv` (line 26) is true, all stages have converged. In the ideal case, every processor has a partial product with rank 1, and thus, the fix up loop executes exactly one iteration. Section 6 shows that we observe the best case for many practical instances.

Say, however, `conv[p]` is not true for processor p. This indicates that the stages $(\mathtt{l_p} \ldots \mathtt{r_p}]$ was not large enough to generate a partial product with rank 1. In the next iteration of the fix up phase, processor $p+1$, in effect, searches for rank convergence in the wider range $(\mathtt{l_p} \ldots \mathtt{r_{p+1}}]$. The fix up loop iteratively combines the stages of the processors till all processors converge. In the worst case, the fix up loop executes $P-1$ iterations and the parallel algorithm devolves to the sequential case.

Important to note is that even though the discussion above refers to partial products, the algorithm does not perform any matrix-matrix multiplications. Like the sequential algorithm, the presentation hides many implementation details in the $\star$ and $\odot$ operations (in lines 10,11,19,and 20). In fact, the parallel implementation can reuse efficient implementations of these operations, including those that use wavefront parallelism, from existing sequential implementations. Also, the computation of `conv` at line 26 is a standard reduce operation that is easily parallelized, if needed.

When compared to the sequential algorithm, the parallel algorithm has to additionally store `s[i]` per stage required to test for convergence in the fix up loop. If space is a constraint, then the fix up loop can be modified to *recompute* `s[i]` in each iteration, trading compute for space.

4.5 All-Non-Zero Invariance

A subtle issue with the correctness of the algorithm above is that starting the LTDP computation midway with an arbitrary initial vector `nz` could produce a zero vector (one with all $\mathbb{0} = -\infty$ entries) at some stage. If this happens, all subsequent stages will produce a zero vector resulting in an erroneous result. To avoid this, we ensure that `nz` is *all-non-zero*, i.e. none of its elements are $\mathbb{0} = -\infty$.

A transformation matrix A is non-trivial, if every row of A contains at least one nonzero entry. In Equation 1, the j row of matrix A_i captures how the subproblem j in stage i depends on the subproblems in stage $i-1$. If *all* entries in this row are $-\infty$, then the subproblem j is forced to be $-\infty$ for any solution to stage $i-1$. Such trivial subproblems can be removed from a given LTDP instance. So, we can safely assume that transformation matrices in LTDP instances are non-trivial.

Lemma 4. *For a non-trivial transformation matrix A,*

$$\vec{v} \textit{ is all-non-zero} \implies A \odot \vec{v} \textit{ is all-non-zero}$$

$(A \odot \vec{v})[i] = max_k(A[i,k] + \vec{v}[k])$. But $A[i,k] \neq -\infty$ for some k ensuring that at least one of the arguments to $\max$ is not $-\infty$. Here we rely on the fact that no element has an inverse under $\max$, except $-\infty$. As such this lemma is not necessarily true in other semirings.

Thus, starting with a all-non-zero vector ensures that none of the stages result in a zero vector.

4.6 Parallel Backward Phase

Once the parallel forward phase is done, performing the sequential backward phase from Figure 2 will generate the right result, even though `s[i]` is not exactly the same as the correct solution $\vec{s}_i$. In many applications, the forward phase overwhelmingly dominates the execution time and parallelizing the backward phase is not necessary. If this is not the case, the backward phase can be parallelized using the same idea as the parallel forward phase as described below.

The backward phase recursively identifies the predecessor at stage i starting from stage n. One way to obtain this predecessor is by iteratively looking up the predecessor products `pred[i]` computed during the forward phase. Another way to obtain this is through repeated matrix multiplication as $M_{i\leftarrow n} \star \vec{s}_i$, where $M_{i\leftarrow n}$ is the backward partial product $A_n \odot \ldots A_{i+1}$. Using the same rank convergence argument, the rank of $M_{i\leftarrow n}$ will converge to 1 for large enough number of matrices (small enough i). Lemma 5 below shows that the predecessor at stages beyond i do not depend on the initial value used for the backward phase.

```
1   Backward_Par(vector pred[1..n]) {
2     vector res;     vector conv;
3     // proc p owns stages (l_p..r_p]
4     ∀p: l_p = n/P*(p-1); r_p = n/P*p;
5     // parallel backward phase
6     parallel.for p in (P..1){
7       // all processors start from 0
8       local x = 0;
9       for i in (r_p..l_p+1) {
10        x = res[i] = pred[i][x]; }}
11    ----- barrier -----
12    do {  // till convergence (fix up loop)
13      parallel.for p in (P-1..1) {
14        conv[p] = false;
15        // obtain final result from next proc
16        local x = res[r_p+1];
17        for i in (r_p..l_p+1) {
18          x = pred[i][x];
19          if (res[i] == x)
20            conv[p] = true;
21            break;                }
22          res[i] = x;               }
23      ----- barrier -----
24      conv = ⋀ conv[p];
               p
25    } while (!conv)
26    return res;                   }
```

Figure 5: Parallel algorithm for the backward phase of LTDP that relies on rank convergence for efficiency. All interprocessor communication is shown in magenta.

Lemma 5. *For a matrix A of rank 1 and any vector $\vec{v}$, all elements of $A \star \vec{v}$ are equal.*

This lemma follows from the fact that the rows in a rank-1 matrix only differ by a constant and $\arg\max$ is invariant when an offset is added to all its arguments.

The algorithm in Figure 4 uses this insight for a parallel backward phase. Every processor starts the predecessor traversal from 0 (line 8) on the stages it owns. Each processor enters a fix up loop whose description and correctness mirror those of the forward phase above.

4.7 Optimizing using Delta Computation

The fix up loop in Figure 4 recomputes solutions `s[i]` for the initial stages for each processor. We have observed that the ranks of the partial products converges to a small rank *much faster* than to rank 1. Intuitively, the old and new values of `s[i]` are almost parallel to each other for these low-rank stages, but still the fix up loop redundantly updates all of their solutions. *Delta* computation optimizes this redundant computation.

Consider parallel vectors $[1, 2, 3, 4]^{\intercal}$ and $[3, 4, 5, 6]^{\intercal}$. Instead, if we represent the vector as the *delta* between adjacent entries along with the first entry, these vectors, represented as $[1, 1, 1, 1]^{\intercal}$ and $[3, 1, 1, 1]^{\intercal}$, are exactly the same except for the first entry. Extending this intuition, if the partial-product at a stage is low-rank, many (but not all) of the entries in the vectors will be the same when represented as deltas. If one modifies the recurrence Equation 1 to operate on deltas, then only the deltas that are different between the old and new values of `s[i]` need to be propagated to the next iteration. This optimization is crucial for instances, such as LCS and Needleman-Wunsch for which the number of solutions in a stage is large and the convergence to low-rank is much faster than the convergence to rank 1.

4.8 Rank Convergence Discussion

One can view solving a LTDP problem as computing shortest/longest paths in a graph. In this graph, each subproblem is a node and directed edges represent the dependences between subproblems. The weights on edges represent the constants $A_i[j, k]$ in Equation 1. In LCS for instance (Figure 1), each subproblem has incoming edges with weight 0 from the subproblem above and to its left, and an incoming edge with weight $\delta_{i,j}$ from its diagonal neighbor. Finding the optimal solution to the LTDP problem amounts to finding the longest path in this graph from the subproblem 0 in the last stage to subproblems in the first stage, given initial weights to the latter. Alternately, one can negate all the weights and change the $\max$ to a $\min$ in Equation 1 to view this as computing shortest paths.

Entries in the partial product $M_{l \to r}$ represent the cost of the shortest (or longest) path from a node in stage l to a node in stage r. The rank of this product is 1 if these shortest paths go through a single node in some stage between l and r. Road networks have this property. For instance, the fastest path from any city in Washington state to any city in Massachusetts is highly likely to go through Interstate I-90 that connects the two states. Routes that use I-90 are overwhelmingly better than those that do not; choices of the cities at the beginning and at the end do not drastically change how intermediate stages are routed. Similarly, if problem instances have optimal solutions that are overwhelmingly better than other solutions, one should expect rank convergence.

5. LTDP Examples

This sections shows four important optimization problems as LTDP — Viterbi, Longest Common Subsequence, Smith-Waterman, and Needleman-Wunsch. Our goal in choosing these particular problems is to provide an intuition on how problems with different structure can be viewed as LTDP. Other problems are LTDP, but not evaluated in this paper, include dynamic time warping and seam carving.

Viterbi The Viterbi algorithm [30] finds the most likely sequence of states in a (discrete) hidden Markov model (HMM) for a given sequence of n observations. Its recurrence equation is shown in Figure 1(a). Here, $p_{i,j}$ represents the probability of the most likely state sequence ending in state j of the HMM that explains the first i observations. The meaning of the term $t_{k,j}$ is not important here (see [30]). The solution to a Viterbi instance is given by the maximum value of $p_{n,j}$ as we are interested in the most likely sequence ending in *any* HMM state.

The subproblems along a column in Figure 1(a) form a stage and they only depend on the subproblems in the previous column. This dependence is not directly in the desired form of Equation 1. But applying logarithm on both sides to the recurrence equation brings it to this form. By transforming the Viterbi instance into one that calculates log-probabilities instead of probabilities, we obtain a LTDP instance.

Invoking the parallel algorithm in Figure 4 requires one additional transformation. The algorithm assumes that the solution to LTDP is given by the first subproblem in the last stage n. To account for this, we introduce an additional stage $n + 1$ in which every subproblem is the maximum of all subproblems in stage n. Essentially, stage $n + 1$ is obtained from multiplying a matrix with 0 in all entries with stage n.

Longest Common Subsequence LCS finds the longest common subsequence of two input strings A and B [12]. The recurrence equation of LCS is shown in Figure 1 (b). Here, $C_{i,j}$ is the length of the longest common subsequence of the first i characters of A and the first j characters of B. Also, $\delta_{i,j}$ is 1 if the ith character of A is the same as the jth character of B and 0 otherwise. The LCS of A and B is obtained by following the predecessors from the bottom-rightmost entry in the table in Figure 1(b).

Some applications of LCS, such as the `diff` utility tool, are only interested in solutions that are at most a width w away from main diagonal - ensuring that the LCS is still reasonably similar to the input strings. For these applications, the recurrence relation can be modified such that $C_{i,j}$ is set to $-\infty$ whenever $|i - j| > w$. Using a smaller width also reduces the memory requirements of LTDP as the entire table need not be stored in memory. Smaller width limits the scope of wavefront parallelism due to smaller sizes of stages, which emphasizes the need for parallelizing across stages as proposed by this paper.

Grouping the subproblems of LCS into stages can be done in two ways, as shown in Figure 6. In the first approach, the stages correspond to anti-diagonals, such as the stage consisting of $\mathtt{z_i}$s in Figure 6 (a). This stage depends on two previous stages (on $\mathtt{x_i}$s and $\mathtt{y_i}$s) and does not strictly follow the rules of LTDP. One way to get around this is to define stages as overlapping pairs of anti-diagonals, like stages

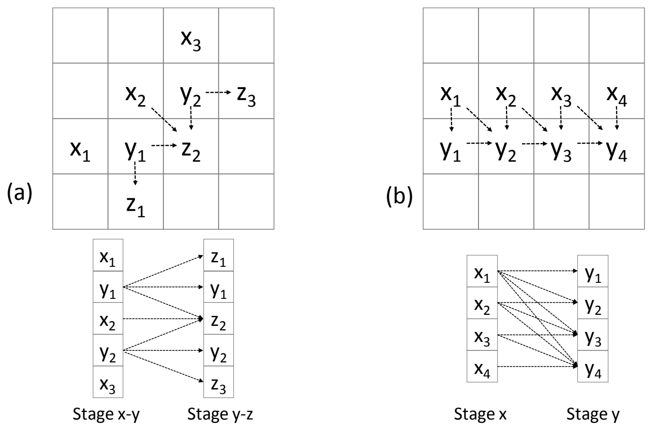


Figure 6: Two ways of grouping the subproblems in LCS into stages such that each stage only depends on one previous stage.

`x-y` and stage `y-z` in Figure 6 (a). Subproblems `yi`s are replicated in both stages, allowing stage `y-z` to depend only on stage `x-y`. While this representation has the downside of doubling the size of each stage, it can sometimes lead to efficient representation. For LCS, one can show that the difference between solutions to consecutive subproblems in a stage is either 1 or 0. This allows compactly representing the stage as a sequence of bits [13]).

In the second approach, the stages correspond to the rows (or columns) as shown in Figure 6 (b). The recurrence needs to be unrolled to avoid dependences between subproblems within a stage. For instance, `yi` depends on all `xj` for `j` $\leq$ `i`. In this approach, since the final solution is obtained from the last entry, the predecessor traversal in Figure 2 has to be modified to start from this entry, say by adding an additional matrix at the end to move this solution to the first solution in the added stage.

Needleman-Wunsch This algorithm [23] finds a *global* alignment of two input sequences, commonly used to align protein or DNA sequences. The recurrence equation is very similar to the one in LCS (Section 5).

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + m[i,j] \\ s_{i-1,j} - d \\ s_{i,j-1} - d \end{cases}$$

In this equation, $s_{i,j}$ is the score of the best alignment for the prefix of length i of the first input and the prefix of length j of the second input, $m[i,j]$ is the matching score for aligning the last characters of the respective prefixes, and d is the penalty for an insertion or deletion during alignment. The base cases are defined as $s_{i,0} = -i * d$ and $s_{0,j} = -j * d$.

Grouping subproblems into stages can done using the same approach as in LCS. Abstractly, one can think of LCS as an instance of Needleman-Wunsch for appropriate values of matching scores and insert/delete penalties. However, the implementation details differ sufficiently enough for us to consider them as two different algorithms.

Smith-Waterman This algorithm [26] performs a *local* sequence alignment, in contrast to Needleman-Wunsch. Given two input strings, Smith-Waterman finds the *substrings* of the input that have the best alignment, where longer substrings have a better alignment. In its simplest form, the recurrence equation is of the form

$$s_{i,j} = \max \begin{cases} 0 \\ s_{i-1,j-1} + m[i,j] \\ s_{i-1,j} - d \\ s_{i,j-1} - d \end{cases}$$

Key difference from Needleman-Wunsch is the 0 term in $\max$ which ensures that alignments “restart” whenever the score goes to zero. Because of this term, the constants in A_i matrices in equation 1 need to be set accordingly. This slight change has significant difference to the convergence properties of Smith-Waterman as we will see later in Section 6.1. Our implementation uses a more complex recurrence equation that allows for affine gap penalties when aligning sequences [8].

Also, the solution to Smith-Waterman requires finding the maximum of all subproblems in all stages and performing a predecessor traversal from that subproblem. To account for this in our LTDP formulation, we add one “running maximum” subproblem per stage that contains the maximum of all subproblems in the current stage and previous stages.

6. Evaluation

This section evaluates the parallel LTDP algorithm on the four problems discussed in Section 5. Section 6.1 empirically evaluates the occurrence of rank convergence in practice. Section 6.3 evaluates scalability, speed up and efficiency of our implementation. Finally, Section 6.4 compares the parallel algorithm with wavefront parallelization.

6.1 LTDP Rank Convergence

Determining whether the LTDP parallel algorithm benefits a dynamic programming problem requires: 1) the problem to be LTDP (discussed in Section 4); 2) rank convergence happens in reasonable number of steps. This section demonstrates how rank convergence can be measured and evaluates it for the 4 LTDP problems discussed in Section 5.

Rank Convergence is an empirical property of a sequence of matrix multiplications that depends on both the LTDP recurrence relation in addition to the input. Table 1 empirically evaluates the number of steps required for rank convergence across different algorithms and inputs. For a LTDP instance, defined by the algorithm (Column 1) and input (Column 2), we first compute the actual solution vectors at each stage. Then, starting from a random all-non-zero vector at 200 different stages, we measured the number of steps required to generate a vector parallel to the actual solution vector (i.e.,

Steps to Converge to Rank-1		Min	Median	Max
Viterbi Decoder	Voyager: 2^6	22	40	104
	LTE: 2^6	18	30	62
	CDMA: 2^8	22	38	72
	MARS: 2^{14}	46	112	414
Smith-Waterman	Query-1: 603	2	6	24
	Query-2: 884	4	8	24
	Query-3: 1227	4	8	24
	Query-4: 1576	4	8	24
Needleman-Wunsch	Width: 1024	1, 580	19, 483	192, 747
	Width: 2048	3, 045	44, 891	378, 363
	Width: 4096	5, 586	101, 085	404, 4374
	Width: 8192	12, 005	267, 391	802, 991
LCS	Width: 8192	9, 142	79, 530	370, 927
	Width: 16384	19, 718	270, 320	–
	Width: 32768	42, 597	626, 688	–
	Width: 65536	86, 393	–	–

Table 1: Number of steps to converge to rank 1.

convergence). Columns 3,4, and 5 respectively show the minimum, median, and maximum number of steps needed for convergence. For each input, Column 2 specifies the computation width (the size of a stage or the size of each A_i matrices). Each algorithm has a specific definition of width: for Viterbi decoder, width is the number of states for each decoder, in Smith-Waterman, it is the size of each query, and in LCS and Needleman-Wunsch, it is a fixed-width around the diagonal of each stage. LCS never converged so we leave those entries blank. The rate of convergence is specific to the algorithm and input (i.e., Smith-Waterman converges fast while LCS sometimes does not converge) and, generally speaking, wider widths require more steps to converge. We will use this table later in Section 6.3 to explain scalability of out approach.

6.2 Environmental Setup

We conducted experiments on a shared memory machine and on a distributed memory machine. A shared memory machine favors fast communication and is ideal for wavefront approach. Likewise, a distributed machine has a larger number of processors and so we can better understand how our parallel algorithm scales. Next, we describe these two machines.

Distributed-Memory Machine: Stampede [27], a Dell PowerEdge C8220 Cluster with 6,400 nodes. At the time of writing this paper, Stampede is ranked 6^{th} on the Top500 [28] list. Each node contains 2 8-core Intel Xeon E5-2600 processor @ 2.70 GHz (16 cores in total) and 32 GB of memory. The interconnect topology is a fat-tree, FDR InfiniBand interconnect. On this cluster, we used the MPI MVAPICH 2 library [21] and the Intel C/C++ compiler version 13.0.1 [14].

Shared-Memory Machine: an unloaded Intel 2.27GHz Xeon (E7) workstation with 40 cores (80 threads with hyperthreading) and 128GB RAM. We use the Intel C/C++ compiler (version 13.0.1) [14] and the Intel MPI library (version 4.1) [15].

We report scalability results on Stampede but results from the shared-memory machine are qualitatively similar. We used the shared-memory machine to compare our parallel algorithm with wavefront parallelization. Unless specified otherwise, the reported results are from Stampede runs.

We use MPI/OMP timer to measure process runtime. We do not measure setup costs — only the time it takes to execute one invocation of a LTDP problem. When we compare against a baseline, we modify that code to take the same measurements.

Finally, to get statistically significant results, we run each experiment multiple times and report the mean and 95% confidence interval of the mean when appropriate. We do not include confidence intervals in the graphs if they are small.

6.3 Parallel LTDP Benchmarks and Performance

This section evaluates the parallel algorithm on four LTDP problems. To substantiate our scalability results, we evaluate each benchmark across a wide variety of real-world inputs. We break the results down by the LTDP problem.

6.3.1 Viterbi Decoder

Viterbi decoder uses the Viterbi algorithm (Section 5) to communicate over noisy and unreliable channels, such as cell phone communications [30]. Given a potentially corrupted convolution-encoded message [24], Viterbi decoding finds the most likely unencoded message.

Baseline We used Spiral's [25] Viterbi decoder: a highly optimized (via auto-tuning) decoder that utilizes SIMD to parallelize decoding within a stage. To the best of our knowledge, there is no efficient multi-processor algorithm for Viterbi decoders since the amount of parallelism in each stage is limited.

Our Implementation Spiral code is heavily optimized and even small changes negatively affect performance. Therefore, the performance-critical internal loop of the Spiral code is used as a black box. Each processor starts from an arbitrary all-non-zero vector (except the first, which uses the initial vector) and uses Spiral to execute its set of stages. Each processor (except the last) then communicates its result to the next processor.

Data We use four real-world convolution codes; Voyager, the convolution codes used on NASA's deep space Voyager. Mars, the convolution codes used to communicate with NASA's mars rovers, and both CDMA and LTE, two convolution codes commonly used in modern cell-phone networks. For each of these 4 convolution codes, we investigate the impact of 4 network packet sizes (2048, 4096, 8192, and 16384), which determine the number of stages in the computation. For each size, we used Spiral's input generator to create 50 network packets.

Performance and Scalability Figure 7 shows the performance, the speed up, and the efficiency of each 4 decoders.

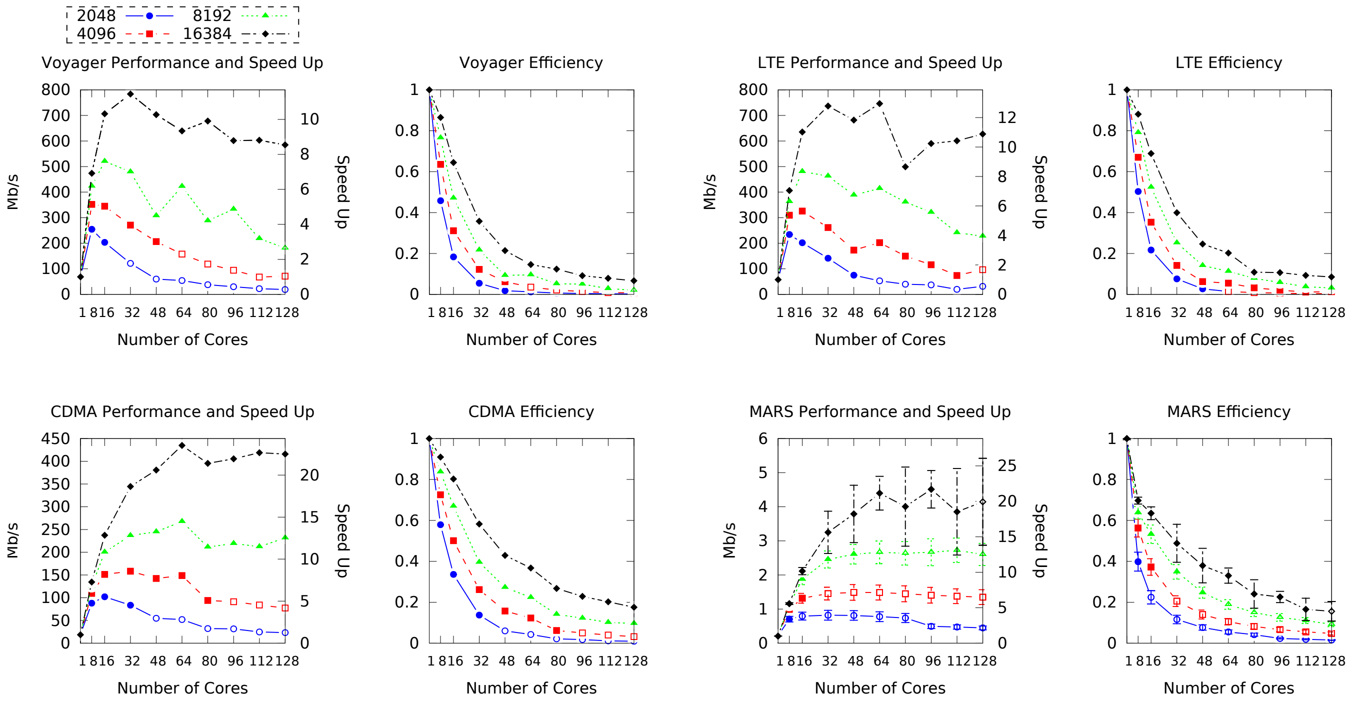


Figure 7: Performance (Mb/S), speed up and efficiency of 4 Viterbi decoders. The non-filled data points demonstrates where processors have too few iterations to converge to rank 1

To evaluate the impact of different decoder sizes, each plot has four lines (one per network packet size). A point (x, y) in a performance/speed up plot with the primary y-axis on left, gives the throughput y (the number of bits processed in a second) in megabits per second (Mb/S) as a function of the number of processors x used to perform the Viterbi Decoding. The same point with the secondary y-axis on right shows the speed up y with x number of processors over the sequential performance. Note that Spiral sequential performance at $x = 1$ is almost the same for different packet sizes. The filled data points in the plots show that convergence occurred in the first iteration of the fix-up loop in Figure 4 algorithm (i.e. each processor's stage is large enough for convergence). The non-filled data points show multiple iterations of the fix-up loop were required. Similarly, a point in an efficiency plot provides the speed up of our parallel implementation over the sequential performance of Spiral generated code divided by the number of processors. Each point is the mean of 50 random packets.

Figure 7 demonstrates (i) our approach provides significant speed ups over the sequential baseline and (ii) different convolution codes and network packet sizes have different performance characteristics. For example, with 64 processors, our CDMA Viterbi Decoder processing packets of size 16384 decodes at a rate of 434 Mb/S which is $24\times$ faster than the sequential algorithm. Note that for the same network packet size and number of processors, our MARS decoder only processes at 4.4 Mb/S because the amount of computation per bit (size of each stage) is significantly greater than CDMA.

The performance of our approach — and thus our speed up numbers — depend on the rate of rank convergence for each pair of convolution codes and network size as shown in Table 1. Larger network packet size provide better performance across all convolution codes (i.e., a network packet size of 16384 is *always* the fastest implementation, regardless of convolution code) because the amount of re-computation (i.e., the part of the computation that has not converged), as a proportion of the overall computation decreases with larger network packet size.

Also, as it can be seen in Figure 7, efficiency plots drop as the packet sizes decrease and this is again because the ratio of the amount of re-computation to the whole computation decreases. Note that with 48 processors, our algorithm for CDMA can reach efficiency of more than 0.4.

6.3.2 Smith-Waterman

As described in Section 5, Smith-Waterman is an algorithm for local sequence alignment [26] often used to align DNA/protein sequences.

Baseline We implemented the fastest known CPU version, Farrar's algorithm, which utilizes SIMD to parallelize within a stage [8].

Our Implementation Our parallel implementation of Smith-Waterman uses Farrar's algorithm as a black-box.

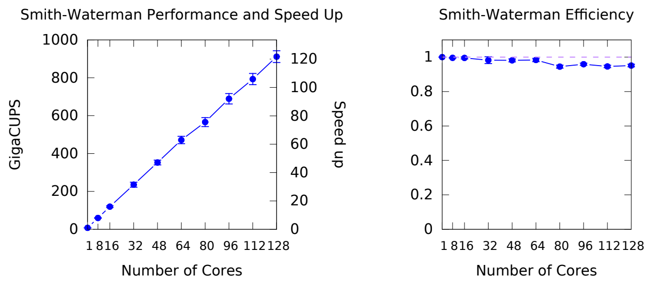


Figure 8: Smith-Waterman performance, speed up and efficiency

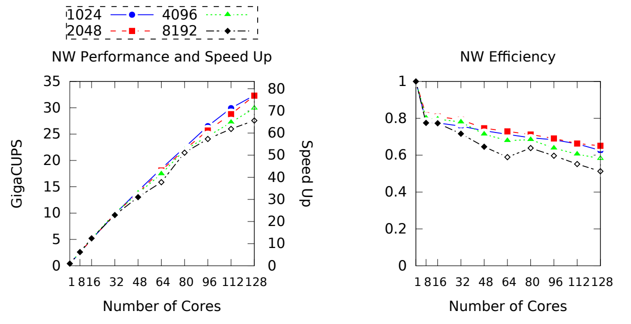
Data We aligned chromosomes 1, 2, 3 and 4 from the human reference genome hg19 as databases and four randomly selected expressed sequence tags as queries. All the inputs are publicly available to download from [22]. We report the average of performance across all combinations of DNA and query (16 in total).

Performance and Scalability A point (x, y) in the performance/speed up plot in Figure 8 with the primary y-axis on left, gives the performance y in Giga cell updates per second, or (GigaCUPS) as a function of the number of processors used to perform the Smith-Waterman alignment. GigaCUPS is a standard metric used in bioinformatics to measure the performance of DNA based sequence alignment problems and refers to the number of cells (in a dynamic programming table) updated per second. Similar to the Viterbi decoder plots, the secondary y-axis on the left show the speed up for each number of processors. We run Smith-Waterman on all combinations of 4 DNA databases and 4 DNA queries (we run each combination 5 times). Unlike the prior Viterbi results, we do not see large variability in performance as a function of the problem data. In other words, the DNA database and query pairs do not significantly impact our performance numbers. This can also be confirmed from Table 1 where the number of steps to converge to rank 1 is significantly smaller than a DNA database size which is more than 100 million long. Thus, we plot the average, across all combinations of DNA databases and queries.

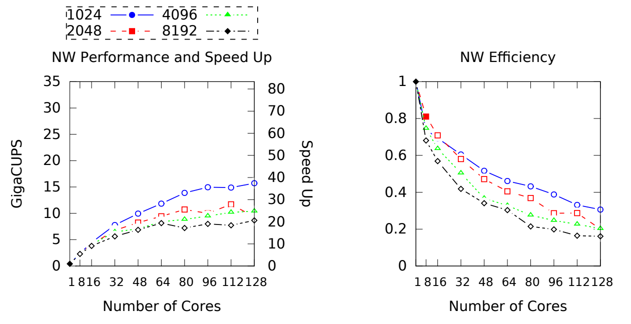
The performance gain of our approach for this algorithm is significant: the efficiency plot in Figure 8 demonstrates that our approach has efficiency $\sim$ 1 for any number of processors which means almost linear speed up with up-to 128 processors. This can be also confirmed from the performance/speed up plot. Our algorithm would scale more with more number of processors but we only report up-to 128 processors to keep Figure 8 consistent with the others.

6.3.3 Needleman-Wunsch

In contrast to Smith-Waterman, which performs a *local* alignment between two sequences, Needleman-Wunsch *globally* aligns two sequences and is often used in bioinformatics to align protein or DNA sequences [23].



(a) Chromosome pair (X, Y): the best performing



(b) Chromosome pair $(21, 22)$: the worst performing

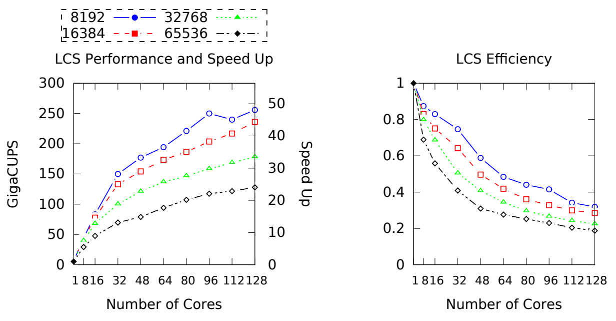
Figure 9: Performance, speed up and efficiency results of Needleman-Wunsch

Baseline We utilized SIMD parallelization *within* a stage for this benchmark by using the grouping technique shown in Figure 6 a.

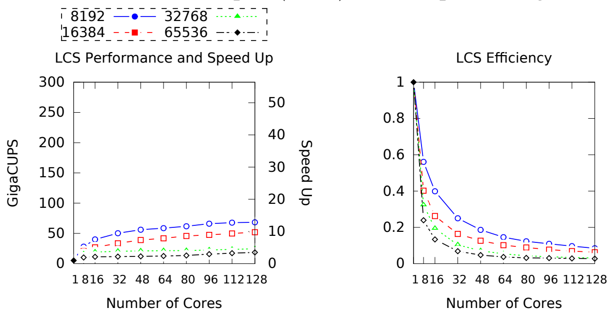
Our Implementation We implemented the incremental optimization described in Section 4.7 using the baseline code.

Data We used 4 pairs of DNA sequences as inputs: Human Chromosomes $(17, 18)$, $(19, 20)$, $(21, 22)$ and (X, Y) from the human reference genome hg19. We only used the first 1 million elements of the sequences since Stampede does not have enough memory on a single node to store the cell values for the complete chromosomes. We also tried 4 different width sizes: 1024, 2048, 4096 and 8192 since we found that widths larger than 8192 do not affect the final alignment score.

Performance and Scalability Figure 9 shows the performance, speed up and efficiency of Needleman-Wunsch algorithm parallelized using our approach for two pairs of chromosomes: (X, Y) and $(21, 22)$. Instead of averaging performance numbers over all 4 pairs, we separated them and reported the best performing pair ((X, Y) in Figure 9a) and the worst performing pair ($(21, 22)$ in Figure 9b). This is because the performance varies significantly between different pairs as can be seen in Figures 9a and 9b. The figures show results for each of the width sizes: 1024, 2048, 4096 and 8192. Similar to the Viterbi decoder benchmark, filled/non-filled data points show whether convergence occurred in the first iteration of the fix up phase.



(a) Chromosome pair (X, Y): the best performing



(b) Chromosome pair $(21, 22)$: the worst performing

Figure 10: Performance, speed up and efficiency results of Longest Common Subsequence

The figures show great variability in performance for different inputs based on the variability in convergence. Also, as it can be seen from non-filled data points and Table 1, rank convergence in this benchmark is not as fast as in Viterbi decoder or Smith-Waterman.

In Figure 9, larger widths perform poorer than smaller ones since the convergence rate depends on the size of each stage in a LTDP instance. Note that we used the same sequence size (1 million element) for all plots.

6.3.4 LCS

Longest Common Subsequence is a method to find the largest subsequence common to two candidate sequences [12] (See Section 5 for description).

Baseline We adapted the fastest known single-core algorithm for LCS that exploits bit-parallelism to parallelize the computation *within* a column [6, 13]. This approach uses the the grouping technique shown in Figure 6 b.

Our Implementation Similar to Needleman-Wunsch, we implemented the incremental optimization described in Section 4.7 using the bit-parallel baseline code.

Data We used the same input data as with Needleman-Wunsch except that we used the following width range: 8192, 16384, 32768 and 65536. We report performance numbers in the same way as in Needleman-Wunsch.

Performance and Scalability The performance, speed up and efficiency plots in Figure 10 are very similar to Figure 9.

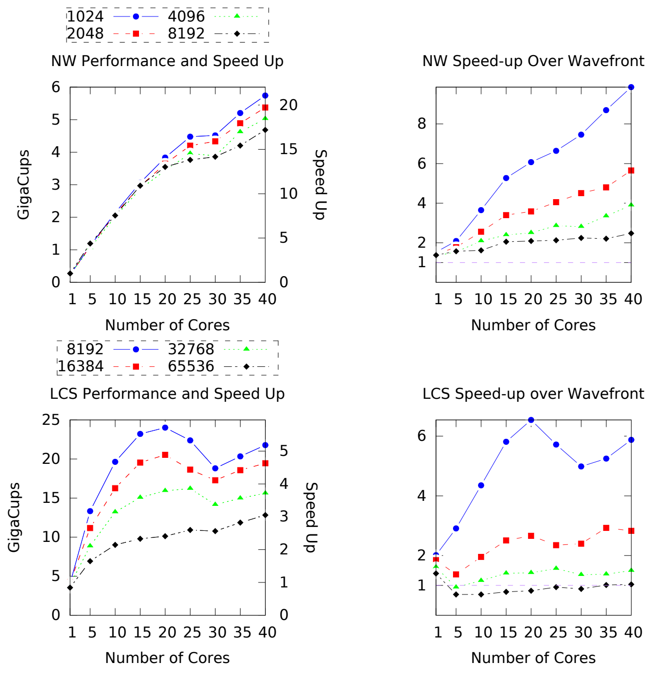


Figure 11: Performance/speed up results and comparison of LTDP and Wavefront for Needleman-Wunsch and LCS

We used the same two pairs of chromosomes: (X, Y) and $(21, 22)$ as they are the best and worst performing pairs respectively. The 4 lines in each plot corresponds to one of following width sizes: 1024, 2048, 4096 and 8192. Likewise, the input pair has a great impact on rank convergence as it can be seen in Figure 10a and Figure 10b.

6.4 Wavefront vs LTDP

Our goal in this section is to directly compare across-stage parallelism with wavefront parallelism. We focus on Needleman-Wunsch and LCS as the size of the stages in Viterbi and Smith-Waterman is very small for wavefront parallelism to be viable. We should note that the two approaches are complementary. Exploring the optimal way to distribute a given budget of processors to simultaneously use across-stage parallelism and within-stage parallelism is left for future work. Furthermore, note that we implemented the best known wavefront algorithm for each of our benchmarks.

We used OpenMP for wavefront implementations and compared it with our MPI implementation used in our Stampede experiments above, but running on our shared-memory machine. This difference in implementation choice should at the worst bias the results against our parallel algorithm.

Wavefront for Needleman-Wunsch: We used tiling to group cells of the computation table and used SIMD in each tile. Wavefronts proceed along the anti-diagonal of these tiles. Tiling greatly reduces the number of barriers involved [19]. On the other hand, processing cells in a tile by utilizing SIMD has computation overhead over the baseline that we used for our parallel approach (without tiling). Therefore, the sequential performance of the baseline with tiling is slower than the baseline without tiling. We investigated different tiling parameters and chose the best performing configuration.

Wavefront for LCS: Similar to the baseline of Needleman-Wunsch, we tiled the cells. For computation in each cell, we used the same bit-parallelism to parallelize the computation within a column of each tile. Likewise, we parallelized computation of tiles that are in the same anti-diagonal.

Figure 11 compares the performance of our approach with an optimized wavefront based approach for both LCS and Needleman-Wunsch. The plots on the left in Figure 11 show the performance and speed up (over sequential non-tiled baseline) of our approach for Needleman-Wunsch and LCS. Plots on the right, a point (x, y) gives the speed up (y as runtime of wavefront divided by runtime of our approach) as we change the number of processors allocated to each approach (x). We plot 4 lines, one for each of four *widths*. Small widths are better for our approach (as wavefront approach incurs more barriers per unit of compute) while large widths are better for wavefront approach (as our approach is less likely to reach rank 1). As we add more processors, our approach utilizes each additional processor more efficiently than a wavefront based approach, particularly so when the width is small (i.e., our approach is $\sim 9\times$ faster than wavefront approach with 40 processors for Needleman-Wunsch and $\sim 6\times$ faster than wavefront approach for LCS with width size 8192).

7. Related Work

There has been a lot of prior work in parallelizing dynamic programming. Predominantly, implementations use wavefront parallelism to parallelize within a stage. In contrast, this paper exploits parallelism across stages in addition to wavefront parallelism. For instance, Martins et al. build a message passing based implementation of sequence alignment dynamic programs (i.e., Smith-Waterman and Needleman-Wunsch) using wavefront parallelism [19]. Our baseline for Needleman-Wunsch builds on this work.

Stivala et al. use an alternate strategy for parallelizing dynamic programming. They use a "top-down" approach that solves the dynamic programming problem by recursively solving the subproblems in parallel. To avoid redundant solutions to the same subproblem, they use a lock-free data structure that memorizes the result of the subproblems. This shared data structure makes it difficult to parallelize across multiple machines.

There is also a large body of theoretical work analyzing the parallel complexity of dynamic programming. Valient et al. [29] show that straight-line programs that compute polynomials in a field, which includes classical dynamic programming, belong to $\mathbf{NC}$, the class of asymptotically efficiently parallelizable problems. Subsequent work [3, 10] has improved both the time complexity and processor complexity of this result. These works view dynamic programming as finding a shortest path in an appropriate grid graph, computing all-pairs shortest paths in partitions of the graph in parallel, and combining the results from each partition efficiently. The works differ in how they use the structure of the underlying graph for efficiency. While it is not clear if these asymptotically efficient algorithms lead to efficient implementation, using the structure of the underlying computation for parallel efficiency is an inspiration for this work.

There are many dynamic programming problem specific implementations. For example, much like we do in this paper, LCS can exploit bit-parallelism (e.g., [1], [5], [13]). And, Aluru et al. describe a prefix-sum approach to LCS[2] which exploits the fact that LCS only uses binary values in its recurrence equation.

Smith-Waterman has been studied extensively due to its importance to DNA sequencing. This paper uses Farrar's SIMD implementation [8] on multi-core, however, prior work has also investigated other hardware (e.g., GPU [18] and FPGA [17]).

Due to its importance in telecommunications, there has been lots of work on parallel Viterbi decoding. Because this algorithm is often implemented in hardware, one simple approach to increase performance is to pipeline via systolic arrays (i.e. to get good throughput) and increase clock frequency (i.e., to get good latency) [9]. The closest approach to us is Fettweis and Meyr who frame Viterbi as linear operations on the tropical semiring and utilize the associativity of matrix-matrix multiplications. However, they suffer linear overheads of this approach which is hidden by adding more hardware.

8. Conclusions

This paper introduces a novel method for parallelizing a class of dynamic programming problems called linear-tropical dynamic programming problems, which includes important optimization problems such as Viterbi and longest-common subsequence. The algorithm uses algebraic properties of the tropical semiring to break data dependence efficiently.

Our implementations show significant speed ups over optimized sequential implementations. In particular, the parallel Viterbi decoding is up-to $24\times$ faster (with 64 cores) than a highly optimized commercial baseline.

While we evaluate our approach on a large shared memory multi-core machine, we expect equally impressive results on a wide variety of parallel hardware platforms (clusters, GPUs and even FPGAs).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS 1111407. The authors thank the Texas Advanced Computing Center for providing computation time on the Stampede cluster. We also thank Serdar Tasiran and anonymous reviewers for useful feedback on the paper.

References

[1] L. Allison and T. I. Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23 (6):305–310, Dec. 1986.

[2] S. Aluru, N. Futamura, and K. Mehrotra. Parallel biological sequence comparison using prefix computations. *J. Parallel Distrib. Comput.*, 63(3):264–272, 2003. ISSN 0743-7315.

[3] A. Apostolico, M. J. Atallah, L. L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990.

[4] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279 – 285, 2001.

[6] S. Deorowicz. Bit-parallel algorithm for the constrained longest common subsequence problem. *Fundamenta Informaticae*, 99(4):409–433, 2010.

[7] M. Develin, F. Santos, and B. Sturmfels. On the rank of a tropical matrix. *Combinatorial and computational geometry*, 52:213–242, 2005.

[8] M. Farrar. Striped Smith-Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

[9] G. Fettweis and H. Meyr. Parallel Viterbi algorithm implementation: breaking the ACS-bottleneck. *IEEE Transactions on Communications*, 37(8):785–790, 1989.

[10] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *Journal of Parallel and Distributed Computing*, 21(2):213–222, 1994.

[11] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

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

[13] H. Hyyro. Bit-parallel LCS-length computation revisited. In *In Proc. 15th Australasian Workshop on Combinatorial Algorithms*, pages 16–27, 2004.

[14] Intel C/C++ Compiler, http://software.intel.com/en-us/c-compilers, 2013.

[15] Intel MPI Library, http://software.intel.com/en-us/intel-mpi-library/, 2013.

[16] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, Oct. 1980.

[17] I. Li, W. Shum, and K. Truong. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA). *BMC Bioinformatics*, 8(1):1–7, 2007.

[18] L. Ligowski and W. Rudnicki. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–8, 2009.

[19] W. S. Martins, J. B. D. Cuvillo, F. J. Useche, K. B. Theobald, and G. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *In Pacific Symposium on Biocomputing*, pages 311–322, 2001.

[20] Y. Muraoka. *Parallelism exposure and exploitation in programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1971.

[21] MVAPICH: MPI over InfiniBand, http://mvapich.cse.ohio-state.edu/, 2013.

[22] National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/, 2013.

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

[24] W. W. Peterson and E. J. Weldon. *Error-Correcting Codes*. MIT Press: Cambridge, Mass, 1972.

[25] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, Special issue on "Program Generation, Optimization, and Adaptation"*, 93:232–275, 2005.

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

[27] *Stampede: Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors*. Texas Advanced Computing Center, http://www.tacc.utexas.edu/resources/hpc.

[28] Top500 Supercompute Sites, http://www.top500.org, 2013.

[29] L. G. Valiant, S. Skyum, S. Berkowitz, and C. Rackoff. Fast parallel computation of polynomials using few processors. *SIAM Journal of Computing*, 12(4):641–644, 1983.

[30] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.