DOI:10.1145/2983553

Efficient Parallelization Using Rank Convergence in Dynamic Programming Algorithms

By Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz

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 speedups 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.

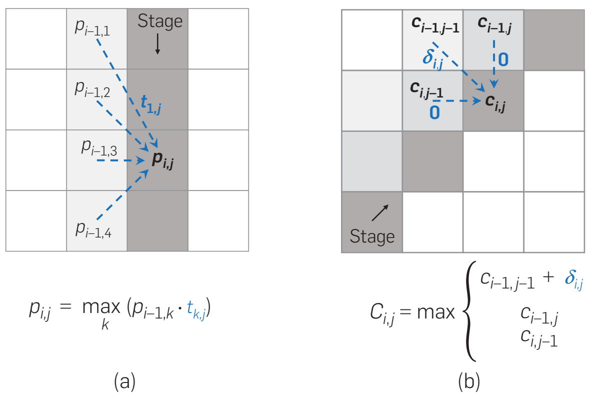
1. INTRODUCTION

Dynamic programming[2] 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,[23] which finds the most-likely path through a hidden-Markov model for a sequence of observations, and LCS,[10] which finds the longest common subsequence between two input strings. Dynamic programming algorithms proceed by recursively solving a series of sub-problems, 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.

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,[15] which computes all subproblems within a stage in parallel.

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

Figure 1. Dynamic programming examples with dependences between stages. (a) the Viterbi algorithm and (b) the LCS algorithm.



Stage
$p_{i-1,1}$
$p_{i-1,2}$
$t_{1,j}$
$p_{i-1,3}$
$p_{i,j}$
$p_{i-1,4}$
$p_{i,j} = \max_k (p_{i-1,k} \cdot t_{k,j})$
(a)
$c_{i-1,j-1}$
$c_{i-1,j}$
$\delta_{i,j}$
0
$c_{i,j-1}$
0
$c_{i,j}$
Stage
$C_{i,j} = \max \begin{cases} c_{i-1,j-1} + \delta_{i,j} \\ c_{i-1,j} \\ c_{i,j-1} \end{cases}$
(b)

with addition as the multiplicative operator and max as the additive operator. This paper demonstrates that several important optimization problems such as Viterbi, LCS, Smith–Waterman, 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[a]. Moreover, the algorithm can reuse existing highly optimized implementations that exploit wavefront parallelism with little modification. As a consequence, our implementation achieves multiplicative speedups over existing implementations. For instance, the parallel Viterbi decoder is up to 24× faster with 64 cores than a state-of-the-art commercial baseline.[18] This paper demonstrates similar speedups for other LTDP instances.

[a] 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 1a.

The original version of this paper is entitled "Parallelizing Dynamic Programming through Rank Convergence" and was published in ACM SIGPLAN Notices – PPoPP '14, August 2014, ACM.

2. BACKGROUND

2.1. Tropical semiring

An important set of dynamic programming algorithms can be expressed in an algebra known as *the tropical semiring*. The tropical semiring has two binary operators: $\oplus$ where $x \oplus y = \max(x, y)$, and $\otimes$ where $x \otimes y = x + y$ for all x and y in the domain. The domain of the tropical semiring is $\{\mathbb{R} \cup \{-\infty\}\}$, the set of real numbers extended with $-\infty$, which serves as the $\mathbb{0}$ of the semiring, meaning that $\mathbb{0} \oplus x = x \oplus \mathbb{0} = x$ and $\mathbb{0} \otimes x = x \otimes \mathbb{0} = \mathbb{0}$. Most properties of ordinary algebra also hold in the tropical semiring, allowing it to support an algebra of matrices over elements of the semiring. For a more detailed discussion of the tropical semiring refer to Section 2 in Ref.[13]

2.2. 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 $A \odot B$, an $l \times n$ matrix defined such that

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

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 $A^{\mathsf{T}}_{m\times n}$ such that $\forall i, j : A^{\mathsf{T}}[i, j] = A[j, i]$. 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 $\vec{v}^{\mathsf{T}}$, and an $x_{1\times 1}$ matrix simply as the scalar x (in which case, by a conventional use of notation, we identify x as $x_{1,1}$). This terminology allows us to extend the definition of matrix–matrix multiplication above to matrix–vector and vector–scalar multiplication appropriately. Also, $\vec{v}[i]$ is the ith element of a vector $\vec{v}$. It is easy to check that matrix multiplication is associative in the tropical semiring: $(A \odot B) \odot C = A \odot (B \odot C)$.

2.3. 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 non-$\mathbb{0}$ scalars x and y such that $\vec{u} \odot x = \vec{v} \odot y$. Intuitively, parallel vectors $\vec{u}$ and $\vec{v}$ in the tropical semiring differ by a constant offset. For instance, $[1\ 0\ 2]^{\mathsf{T}}$ and $[3\ 2\ 4]^{\mathsf{T}}$ are parallel vectors differing by an offset 2.

2.4. Matrix rank

The rank of a matrix $M_{m\times n}$, denoted by 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 ordinary linear algebra, they are not equivalent in arbitrary semirings.[4])

LEMMA 1. *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 rank(A) = 1 then it is the product of a column vector $\vec{c}$ and a row vector $\vec{r}^{\mathsf{T}}$. For any vectors $\vec{u}$ and $\vec{v}$:

$$A \odot \vec{u} = \left(\vec{c} \odot \vec{r}^{\mathsf{T}}\right) \odot \vec{u} = \vec{c} \odot \left(\vec{r}^{\mathsf{T}} \odot \vec{u}\right) = \vec{c} \odot x_u$$
$$A \odot \vec{v} = \left(\vec{c} \odot \vec{r}^{\mathsf{T}}\right) \odot \vec{v} = \vec{c} \odot \left(\vec{r}^{\mathsf{T}} \odot \vec{v}\right) = \vec{c} \odot x_v$$

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

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

$A = [1\ 2\ 3]^{\mathsf{T}} \odot [0\ 1\ 2]$ is rank-1. $A \odot \vec{u} = [6\ 7\ 8]^{\mathsf{T}}$ and $A \odot \vec{v} = [4\ 5\ 6]^{\mathsf{T}}$ 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.

3.1. LTDP definition

A dynamic programming problem is LTDP if (a) its solution and the solutions of its subproblems are in the domain of the tropical semiring, (b) the subproblems can be grouped into a sequence of stages such that the solution to a subproblem in a stage depends on only the solutions in the previous stage, and (c) 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

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

for appropriate constants $A_i[j, k]$. This linear dependence allows us to view LTDP as computing, from an initial solution vector $\vec{s}_0$ (obtained from the base case for the recurrence equation), a sequence of vectors $\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n$, where the vectors need not have the same length, and

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

for appropriate matrices of constants A_i derived from the recurrence equation. We will call $\vec{s}_i$ the *solution vector* at stage i and call A_i the *transformation matrix* at stage i.

3.2. Backward phase

Once all the subproblems are solved, finding the solution to the underlying LTDP optimization problem usually involves tracing the *predecessors* of subproblems backward. 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

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

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 2. $\vec{u} \parallel \vec{v} \Rightarrow \forall A : A \star \vec{u} = A \star \vec{v}$

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.

3.3. Sequential LTDP

The *sequential* algorithm for LTDP can be phrased in terms of matrix multiplications using Equations 2. Assume that the LTDP problem has $n+1$ stages $\vec{s}_0, \vec{s}_1, \ldots, \vec{s}_n$ where $\vec{s}_0$ is the initial solution vector given as an input argument and $\forall i : 1 \leq i \leq n, \vec{s}_i$ are the desired output solutions. The output solutions can be computed using the transformation matrices $A_1, \ldots, A_n$ which capture the inductive case of the LTDP recurrence as shown in Equation (2). The sequential algorithm consists of a forward phase and a backward phase.

The forward phase computes the solution iteratively in n iterations. In iteration i, it computes $\vec{s}_i = A_i \odot \vec{s}_{i-1}$ and $\vec{p}_i = A_i \star \vec{s}_{i-1}$. This algorithm is deemed sequential because it computes the stages one after the other due to the data-dependence across stages.

The backward phase iteratively follows the predecessors of solutions computed in the forward phase. Depending on the LTDP problem, one of the subproblems in the last stage, say $s_n[opt]$, contains the optimal solution. Then the solutions along the optimal path for the LTDP problem are

$$\begin{gathered} s_n[opt], \\ s_{n-1}[p_n[opt]], \\ s_{n-2}[p_{n-1}[p_n[opt]]], \\ \ldots, \\ s_1[p_2[p_3[\ldots[p_{n-1}[p_n[opt]]]\ldots]]], \\ s_0[p_1[p_2[p_3[\ldots[p_{n-1}[p_n[opt]]]\ldots]]]] \end{gathered}$$

These are easily computed by in linear time by tracing the path backwards from stage n to stage 0.

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 (dense) vector and perform (dense) matrix–vector operations. It might statically know that the current solution depends on only some of the subproblems in the previous stage (a sparse matrix) and only accesses those. Moreover, as mentioned above, an implementation might use wavefront parallelism to compute the solutions in a stage in parallel. Finally, implementations can use techniques such as tiling to improve the cache-efficiency of the sequential computation. 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 parallel algorithm for the sequential algorithm described in Section 4 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

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

Standard techniques[9, 12] can parallelize this computation using the associativity of matrix multiplication. For instance, two processors can compute the partial products $A_{hi} = A_n \odot \cdots \odot A_{\lfloor n/2 \rfloor + 1}$ and $A_{lo} = A_{\lfloor n/2 \rfloor} \odot \cdots \odot A_1$ in parallel, and then compute $A_{hi} \odot A_{lo} \odot \vec{s}_0$ to obtain $\vec{s}_n$.

However, doing so converts a sequential computation that performs matrix–vector multiplications (working from right to left) 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 a linear number of processors to observe a constant speedup. In practice, the size of each stage 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 algorithm relies on the convergence of matrix rank in the tropical semiring as discussed below. Its exposition requires the following definition: For a given LTDP instance, the *partial product* $M_{i \to j}$, defined for stages $i \leq j$, is given by

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

Note that $M_{i \to j}$ describes how stage j depends on stage i, because $\vec{s}_j = M_{i \to j} \odot \vec{s}_i$.

4.2. Rank convergence

The rank of the product of two matrices is not greater than the rank of either input matrix.

$$\text{rank}\,(A \odot B) \leq \min\left(\text{rank}\,(A), \text{rank}\,(B)\right) \tag{3}$$

This is because, if 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 rank($A \odot B$) $\leq$ rank(A). A similar argument shows that rank($A \odot B$) $\leq$ rank(B).

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

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

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 observed this only 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 1, 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; stage i determines only its magnitude. In the tropical semiring, where the multiplicative operator is +, this means that the solution vector at stage j will be, at worst, off by a constant if one starts stage i with an arbitrary vector.

4.3. Parallel forward phase

The parallel algorithm uses this insight to break dependences between stages, as shown pictorially in Figure 2. The figure uses three processors, P_0, P_1, and P_2, and six stages for each processor as an example, for a total of 18 stages beyond $\vec{s}_0$. Figure 2a represents the forward phase of the sequential algorithm described in Section 4. Each stage is represented as a vertical column of cells. (For pictorial simplicity, we assume each solution vector has length 4, but in general they might have different lengths.) Note that P_0 also contains the initial solution $\vec{s}_0$; note also that stage $\vec{s}_6$ is shared between P_0 and P_1, and similarly stage $\vec{s}_{12}$ is shared between P_1 and P_2. These replicated stages are differentiated by dotted borders. Each stage $\vec{s}_i$ is computed by multiplying $\vec{s}_{i-1}$ by the transformation matrix A_i (Equation 2). Processor P_0 starts from the initial solution vector s_0 and computes all its stages. As indicated by the arrow on the left, processor P_1 waits for P_0 to compute the shared stage $\vec{s}_6$ in order to start its computation. Similarly, processor P_2 waits for P_1 to compute the shared stage $\vec{s}_{12}$ as the arrow on the right shows.

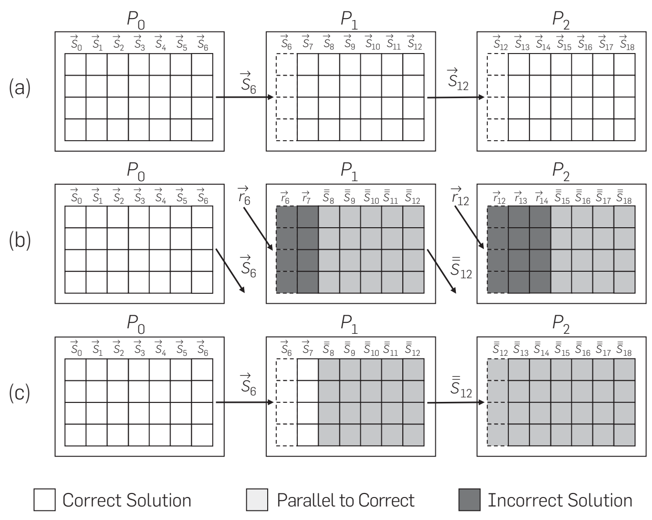
In the parallel algorithm shown in Figure 2b, processors P_1 and P_2 start from *arbitrary* solutions $\vec{r}_6$ and $\vec{r}_{12}$ respectively in parallel with P_0. Of course, the solutions for the stages computed by P_1 and P_2 will start out as completely wrong (shaded dark in the figure). However, 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_1 will generate some solution vector $\bar{\bar{s}}_{12}$ parallel to $\vec{s}_{12}$.

In a subsequent *fixup phase*, shown in Figure 2c, P_1 uses $\vec{s}_6$ computed by P_0, and P_2 uses $\bar{\bar{s}}_{12}$ computed by P_1, to fix stages that are not parallel to the actual solution vector at that stage. The fixup phase terminates when a newly computed stage is parallel to (off by a constant from) the computed one from the forward phase. (If rank convergence had not occurred with P_1, then in the fixup phase, P_1 might have had to fix all of its stages including $\vec{s}_{12}$ since it is not parallel to the $\vec{s}_{12}$ it had computed from $\vec{r}_6$. In such a case, P_2 would need to re-perform its fixup phase. Thus, in the worst case, the parallel algorithm reduces gracefully to the serial algorithm.) After the fixup, 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. Because parallel vectors generate the same predecessor products (Lemma 2), following the predecessors in Figure 2c will generate the same solution as the following the predecessors in Figure 2a.

A general version of this algorithm with pseudocode is discussed in Section 4 of Ref.[13] For correctness (see Section 4.4 in Ref.[13]), the parallel algorithm requires that every entry in the arbitrary vectors ($\vec{r}_6$ and $\vec{r}_{12}$ in Figure 2) be non-$\mathbb{0}$. The backward phase can also be parallelized using ideas similar to those used in the forward phase (see Section 4.6 in Ref.[13]).

Figure 2. Parallelization algorithm using rank convergence. (a) the sequential forward phase, (b) the parallel forward phase, and (c) the fixup phase.



P_0
P_1
P_2
(a)
(b)
(c)
$\vec{s}_0$ $\vec{s}_1$ $\vec{s}_2$ $\vec{s}_3$ $\vec{s}_4$ $\vec{s}_5$ $\vec{s}_6$
$\vec{s}_6$ $\vec{s}_7$ $\vec{s}_8$ $\vec{s}_9$ $\vec{s}_{10}$ $\vec{s}_{11}$ $\vec{s}_{12}$
$\vec{s}_{12}$ $\vec{s}_{13}$ $\vec{s}_{14}$ $\vec{s}_{15}$ $\vec{s}_{16}$ $\vec{s}_{17}$ $\vec{s}_{18}$
$\vec{r}_6$ $\vec{r}_7$ $\bar{\bar{s}}_8$ $\bar{\bar{s}}_9$ $\bar{\bar{s}}_{10}$ $\bar{\bar{s}}_{11}$ $\bar{\bar{s}}_{12}$
$\vec{r}_{12}$ $\vec{r}_{13}$ $\vec{r}_{14}$ $\bar{\bar{s}}_{15}$ $\bar{\bar{s}}_{16}$ $\bar{\bar{s}}_{17}$ $\bar{\bar{s}}_{18}$
$\vec{r}_6$
$\vec{r}_{12}$
$\vec{s}_6$
$\vec{s}_{12}$
$\bar{\bar{s}}_{12}$
□ Correct Solution
▨ Parallel to Correct
■ Incorrect Solution

4.4. Rank convergence discussion

One can view solving a LTDP problem as computing shortest (or 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 (Figure 1b), each subproblem has incoming edges with weight 0 from the subproblems above it 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 $C_{0,0}$ to $C_{m,n}$ (where m and n are the lengths of the two input strings). Alternatively, 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 all 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 Chicago, because routes that use Interstate 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 section shows LTDP solutions for four important optimization problems: 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 applications for LTDP, not evaluated in this paper, include dynamic time warping and seam carving.

5.1. Viterbi

The Viterbi algorithm[23] 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 1a (refer to Ref.[23] for the meaning of the $p_{i,j}$ and $t_{k,j}$ terms). The subproblems along a column in the figure form a stage and they only depend on subproblems in the previous column. This dependence is not directly in the desired form of Equation (1), but applying the logarithm function to both sides of 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.

5.2. Longest common subsequence

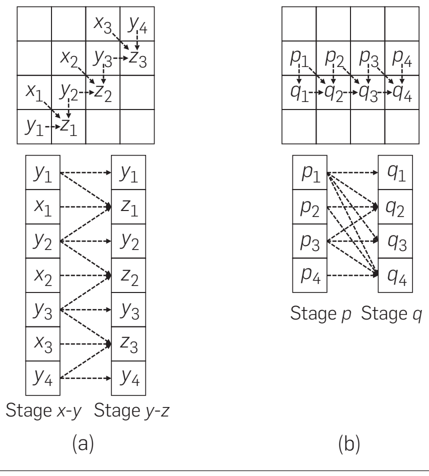
LCS finds the longest common subsequence of two input strings A and B.[10] The recurrence equation of LCS is shown in Figure 1b. Here, $C_{i,j}$ is the length of the longest common subsequence of the first i characters of A and the first j characters of B (where adjacent characters of a subsequence need not be adjacent in the original sequence, but must appear in the same order). 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 1b.

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 $|i - j| > w$. In effect, this modification limits the size of each stage $\vec{s}_i$, which in turn limits wavefront parallelism, increasing the need to execute multiple stages in parallel as we propose here.

Grouping the subproblems of LCS into stages can be done in two ways, as shown in Figure 3. In the first approach, the stages correspond to anti-diagonals, such as the stage consisting of z_is in Figure 3a. This stage depends on two previous stages (on x_is and y_is) 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 stage x–y and stage y–z in Figure 3a. Subproblems y_is 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.[11]

In the second approach, the stages correspond to the rows (or columns) as shown in Figure 3b. The recurrence needs to be unrolled to avoid dependences between subproblems within a stage. For instance, q_i depends on all p_j for $j \le i$. In this approach, since the final solution is obtained from the last entry, the predecessor traversal in the backward phase 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.

Figure 3. Two ways of grouping the subproblems in LCS into stages so that each stage depends on only one previous stage. (a) anti-diagonal grouping and (b) row grouping.



5.3. Needleman–Wunsch

This algorithm[17] 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.

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

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 $s_{i,0} = -i \cdot d$ and $s_{0,j} = -j \cdot d$. Also, grouping subproblems into stages can done using the same approach as in LCS.

5.4. Smith–Waterman

This algorithm[19] 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

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

The key difference from Needleman–Wunsch is the 0 term in max which ensures that alignments "restart" whenever the score goes below zero. Because of this term, the constants in the A_i matrices in Equation 1 need to be set accordingly. This slight change significantly affects the convergence properties of Smith–Waterman (see Section 6.1).

6. EVALUATION

This section evaluates the parallel LTDP algorithm on the four problems discussed in Section 5.

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) and (2) rank convergence to happen in a reasonable number of steps. This section demonstrates how rank convergence can be measured and evaluates it for the 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 and the input. Table 1 presents measurements of 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 equally spaced stages, we measured the number of steps required to converge to a vector parallel to the actual solution vector. 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 each stage). 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 in some cases never converged, so we left those entries blank. The rate of convergence is specific to the algorithm and input (e.g., Smith–Waterman converges fast while LCS sometimes does not converge) and, generally speaking, wider widths require more steps to converge. We use this table later in Section 6.3 to explain scalability of our 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 the wave-front approach. The distributed-memory machine has a larger number of processors, so we can better understand how our parallel algorithm scales. The shared-memory machine has 40 cores (Intel Xeon E7). The distributed-memory machine is called Stampede[21]; for our experiments we used up to 128 cores (Intel Xeon E5). See Ref.[13] for more details.

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. For each LTDP problem, we used the best existing sequential algorithm as our baseline, as described below. For the parallel LTDP algorithm, we implemented the algorithm described in Section 4 and used the sequential baseline implementation as a black box to advance from one stage to the next.

Viterbi decoder. As our baseline, we used Spiral's[18] Viterbi decoder, a highly optimized (via auto-tuning) decoder that utilizes SIMD to parallelize decoding within a stage. We use two real-world convolution codes, CDMA and LTE, which are commonly used in modern cell-phone networks. For each of these two convolution codes, we investigated the impact of four network packet sizes (2048, 4096, 8192, and 16,384), which determine the number of stages in the computation. For each size, we used Spiral's input generator to create 50 network packets and considered the average performance.

Figure 4 shows the performance, the speedup, and the efficiency of the two decoders. 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/speedup 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 speedup 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 indicate that convergence occurred in the first iteration of the fixup phase in the parallel LTDP algorithm described in Section 4 (i.e., each processor's stage is large enough for convergence). The non-filled data points indicate that multiple iterations of the fixup loop were required.

Figure 4 demonstrates that (i) our approach provides significant speedups 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

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

Steps to converge to Rank-1		Min	Median	Max
Viterbi decoder	LTE: 2^6	18	30	62
	CDMA: 2^8	22	38	72
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	1580	19,483	192,747
	Width: 2048	3045	44,891	378,363
	Width: 4096	5586	101,085	404,437
	Width: 8192	12,005	267,391	802,991
LCS	Width: 8192	9142	79,530	370,927
	Width: 16,384	19,718	270,320	–
	Width: 32,768	42,597	626,688	–
	Width: 65,536	86,393	–	–

of size 16,384 decodes ~ 24× faster than the sequential algorithm while this number is ~ 13× for our LTE decoder. This difference is due to the fact that the amount of computation per bit in CDMA is four times as much as in LTE but the median of the convergence rate is almost the same (Table 1). Also, note that in Figure 4, larger network packet size provide better performance across all convolution codes (i.e., a network packet size of 16,384 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.

Smith–Waterman. Our baseline implements the fastest known CPU version, Farrar's algorithm, which utilizes SIMD to parallelize within a stage.[5] For 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 Ref.[16] We reported the average of performance across all combinations of DNA and query (16 in total).

A point (x, y) in the performance/speedup plot in Figure 5 with the primary y-axis on left, gives the performance y in giga cell updates per second (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 speedup for each number of processors.

Figure 4. Performance (Mb/S) and speedup of two Viterbi decoders. The non-filled data points indicate where processors have too few iterations to converge to rank 1

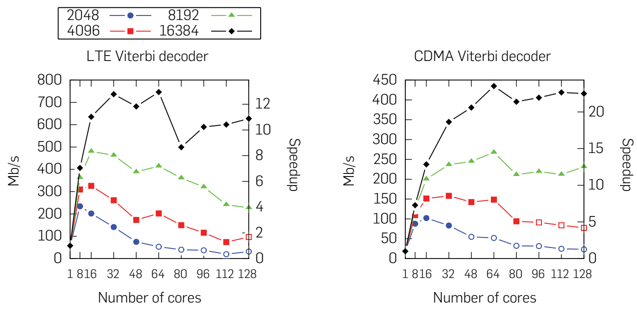
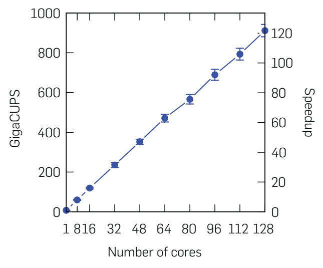


Figure 5. Smith–Waterman performance and speedup averages over four databases and four queries.



The performance gain of our approach for this algorithm is significant. As Figure 5 demonstrates, the parallel LTDP algorithm delivers almost linear speedup with up to 128 processors. Our algorithm scales well with even higher numbers of processors, but we report only up to 128 processors to keep Figure 5 consistent with the others.

Needleman–Wunsch. Our baseline utilized SIMD parallelization *within* a stage by using the grouping technique shown in Figure 3a. For data, we used two pairs of DNA sequences as inputs: Human Chromosomes (21, 22) and (X, Y) from the human reference genome hg19. We used only the first 1 million elements of each sequence since Stampede does not have enough memory on a single node to store the cell values for the complete chromosomes. We also used just 4 different widths (1024, 2048, 4096, and 8192) since we found that widths larger than 8192 do not affect the final alignment score.

Figure 6 shows the performance and speedup of the Needleman–Wunsch algorithm parallelized using our approach. The rank convergence is dependent on the input data; consequently, the parallel LTDP algorithm performs differently with pairs (X, Y) and (21, 22), as can be seen in the figure. The plots in the figure 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 fixup phase.

In Figure 6, larger widths perform worse than smaller ones since the convergence rate depends on the size of each stage in a LTDP instance.

LCS. Our baseline adapts the fastest known single-core algorithm for LCS that exploits bit-parallelism to parallelize the computation *within* a column.[3, 11] This approach uses the grouping technique shown in Figure 3b. For data, we used the same input data as with Needleman–Wunsch except that we used the following 4 widths: 8192, 16,384, 32,768, and 65,536. We report performance numbers as for Needleman–Wunsch.

The performance and speedup plots in Figure 7 are very similar to those in Figure 6: The choice of input pair has a great impact on rank convergence, as can be seen in Figure 7.

LTDP versus Wavefront. We have also compared our LTDP parallel approach and the wavefront approach for LCS and Needleman–Wunsch. Refer to Section 6.4 in Ref.[13] for details.

Figure 6. Performance and speedup of Needleman–Wunsch.

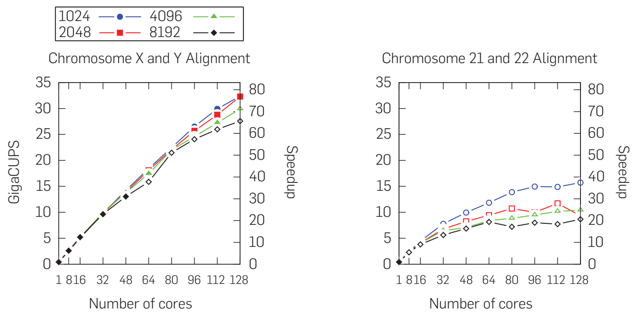
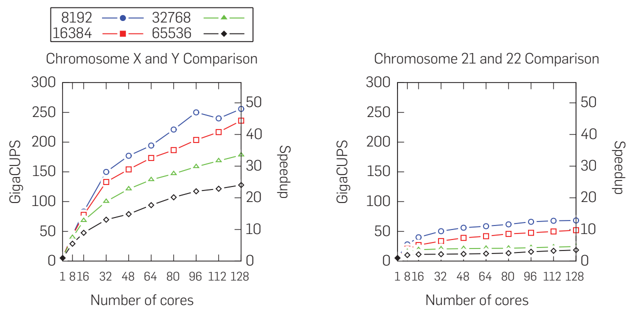


Figure 7. Performance and speedup results of Longest Common Subsequence.



7. RELATED WORK

Due to its importance, there is a lot of prior work on parallelizing dynamic programming. Predominantly, implementations use wavefront parallelism to parallelize within a stage. 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.[14] In contrast, this paper exploits parallelism across stages, which is orthogonal to wave-front parallelism.

Stivala et al. use an alternate strategy for parallelizing dynamic programming.[20] 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 memoizes 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 on the parallel complexity of instances of dynamic programming. Some of them[1, 8, 22] view dynamic programming instances as finding a shortest path in an appropriate graph and compute all-pairs shortest paths in graph partitions in parallel. Our work builds on these insights and can be viewed as using rank convergence to compute the all-pairs shortest paths efficiently.

Prior works have also made and utilized observations similar to rank convergence. The classic work on Viterbi decoding[24] uses the convergence of decoding paths to synchronize decoding and to save memory by truncating paths for the backward phase. Fettweis and Meyr[6,7] use this observation to parallelize Viterbi decoding by processing overlapping chunks of the input. However, their parallelization can produce an erroneous decoding, albeit under extremely rare conditions.

8. CONCLUSION

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 speedups over optimized sequential implementations. In particular, the parallel Viterbi decoding is up to 24× faster (with 64 cores) than a highly optimized commercial baseline.

While we evaluate our approach on a cluster, we expect equally impressive results on a variety of parallel hardware platforms (shared-memory machines, GPUs, and 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 greatly thank Guy Steele for his invaluable comments and efforts for editing this paper. We also extend our thanks to Serdar Tasiran and anonymous reviewers for useful feedback on the paper.

References

1. Apostolico, A., Atallah, M.J., Larmore, L.L., McFaddin, S. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput. 19*, 5 (1990), 968–988.
2. Bellman, R. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.
3. Deorowicz, S. Bit-parallel algorithm for the constrained longest common subsequence problem. *Fund. Inform. 99*, 4 (2010), 409–433.
4. Develin, M., Santos, F., Sturmfels, B. On the rank of a tropical matrix. *Combin. Comput. Geom. 52* (2005), 213–242.
5. Farrar, M. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics 23*, 2 (2007), 156–161.
6. Fettweis, G., Meyr, H. Feedforward architectures for parallel Viterbi decoding. *J. VLSI Signal Process. Syst.* 3, 1–2 (June 1991), 105–119.
7. Fettweis, G., Meyr, H. High-speed parallel Viterbi decoding: algorithm and VLSI-architecture. *Commun. Mag. IEEE 29*, 5 (1991), 46–55.
8. Galil, Z., Park, K. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *J. Parallel Distrib. Comput. 21*, 2 (1994), 213–222.
9. Hillis, W.D., Steele, G.L., Jr. Data parallel algorithms. *Commun. ACM 29*, 12 (Dec. 1986), 1170–1183.
10. Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences. *Commun. ACM 18*, 6 (June 1975), 341–343.
11. Hyyro, H. Bit-parallel LCS-length computation revisited. In *Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms* (2004), 16–27.
12. Ladner, R.E., Fischer, M.J. Parallel prefix computation. *J. ACM 27*, 4 (Oct. 1980), 831–838.
13. Maleki, S., Musuvathi, M., Mytkowicz, T. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14 (New York, NY, USA, 2014). ACM, 219–232.
14. Martins, W.S., Cuvillo, J.B.D., Useche, F.J., Theobald, K.B., Gao, G. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Pacific Symposium on Biocomputing* (2001), 311–322.
15. Muraoka, Y. Parallelism exposure and exploitation in programs. PhD thesis, University of Illinois at Urbana-Champaign (1971).
16. National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov/. 2013.
17. Needleman, S.B., Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol. 48* (1970) 443–453.
18. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE, Special Issue on "Program Generation, Optimization, and Adaptation" 93* (2005) 232–275.
19. Smith, T., Waterman, M. Identification of common molecular subsequences. *J. Mol. Biol. 147*, 1 (1981), 195–197.
20. Stivala, A., Stuckey, P.J., Garcia de la Banda, M., Hermenegildo, M., Wirth, A. Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput. 70*, 8 (Aug. 2010), 839–848.
21. Texas Advanced Computing Center, http://www.tacc.utexas.edu/resources/hpc. *Stampede: Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors.*
22. Valiant, L.G., Skyum, S., Berkowitz, S., Rackoff, C. Fast parallel computation of polynomials using few processors. *SIAM J. Comput. 12*, 4 (1983), 641–644.
23. Viterbi, A. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inform. Theory 13*, 2 (1967), 260–269.
24. Viterbi, A.J., Omura, J.K. *Principles of Digital Communication and Coding.* Communications and information theory. McGraw-Hill, New York, 1979. Autre réimpr.: 1985.

Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz ([saemal, madanm, toddm]@microsoft.com), Microsoft Research, Redmond, WA.