Parallel Algorithms for Fast Computation of Normalized Edit Distances (Extended Abstract)

Ömer Eğecioğlu * and *Maximilian Ibel*

Department of Computer Science
University of California Santa Barbara, CA 93106
{omer,ibel}@cs.ucsb.edu

Abstract

We give work-optimal and polylogarithmic time parallel algorithms for solving the normalized edit distance problem. The normalized edit distance between two strings X and Y with lengths $n \geq m$ is the minimum quotient of the sum of the costs of edit operations transforming X into Y by the length of the edit path corresponding to those edit operations. Marzal and Vidal proposed a sequential algorithm with a time complexity of $O(nm^2)$. We show that this algorithm can be parallelized work-optimally on an array of n (or m) processors, and on a mesh of $n \times m$ processors. We then propose a sublinear time algorithm that is almost work-optimal: using $O(mn^{1.75})$ processors, the time complexity of the algorithm is $O(n^{0.75} \log n)$ and the total number of operations is $O(mn^{2.5} \log n)$. This algorithm runs on a CREW PRAM, but is likely to work an weaker PRAM models and hypercubes with minor modifications. Finally, we present a polylogarithmic $O(\log^2 n)$ time algorithm based on matrix multiplication which runs on a $O(n^6 / \log n)$ processor hypercube.

Keywords*: Dynamic programming, edit distance, normalized edit distance, parallel algorithm, matrix multiplication, fractional programming, information retrieval, pattern matching.*

1. Introduction

Categorizing and comparing strings over a finite alphabet is an important task in such diverse fields as text processing, OCR, image and signal processing, error correction, information retrieval, pattern recognition, and pattern matching in large databases.

To compare the similarity of two strings X and Y with lengths n and m respectively, with $n \geq m$, we can define the *edit distance* of X and Y as the minimum weight of a sequence of weighted edit operations (i.e. substitutions, insertions, or deletions of characters), transforming X into Y. Although the edit distance provides a useful tool for determining the closeness of two strings, it suffers from some disadvantages. First, the edit distance does not take the lengths of the compared strings into account. For example, two binary string of length 1000 differing in 1 bit have the same edit distance as two binary strings of length 2 differing in 1 bit, although one would most likely state that only the 1000-bit strings are "almost equal". A logical step to overcome this shortcoming is to post-normalize the edit distance, that is, first compute the ordinary edit distance between X and Y, and then divide the obtained distance by the number of edit operations used. However, this approach introduces several deficiencies: For one thing, we now do not only have to know the edit distance, but rather an optimal sequence itself so that we can determine its length. More importantly, it is possible to have two sequences of editing transformations with the same minimal cost but different lengths, so that the post-normalization is not even well-defined without further restrictions. Finally, the post-normalized edit distance does not satisfy the triangle inequality [8].

In order to overcome the disadvantages of the regular or post-normalized edit distance, Marzal and Vidal define the *normalized edit distance* of two strings X and Y by considering *editing paths* between X and Y [8]. The length $L(P)$ of an editing path P lies in the range $n \leq L(P) \leq n + m$. If $W(P)$ denotes the weight of P, then the normalized edit distance between X and Y is defined to be the minimum of the quotients $W(P)/L(P)$, as P runs through all editing paths between X and Y. The normalized edit distance is well-defined, satisfies the triangle inequality, and more importantly yields better results in empirical experiments

*Supported in part by NSF/NASA/ARPA Grant No. IRI94-11330.

[8]. Note that the normalized edit distance does not, as one might think, always favor longer request sequences, since the edit sequence length $L(P)$ does not grow independently of the weight of $W(P)$.

Unfortunately, the computation of the normalized edit distance requires more work than that of the conventional edit distance. Whereas the conventional edit distance can be computed in $O(mn)$ sequential time [11, 14, 2] (or $O(mn/\log n)$ if the weights are rational, [9]), the best bound for the normalized edit distance calculation is $O(nm^2)$ [8] [1].

Due to the high computing cost of the normalized edit distance, it makes sense to investigate means of parallelizing this computation. There are various algorithms in the literature for the parallelization of the ordinary edit distance problem, for example [4, 10]. In this paper we provide several different parallel algorithms for the normalized edit distance problem. First, we present a work-optimal algorithm running on an array of up to n processors, and also a work-optimal algorithm on a mesh of $n \times m$ processors. These two algorithms have optimal speedup and exploit the regularity of the dynamic programming formulation for the normalized edit distance problem. Similar approaches have been undertaken for the conventional edit distance in the literature [1, 6]. Adapting a shortcut developed by Larmore and Rytter [7], we then present a sublinear time algorithm with complexity $O(n^{0.75}\log n)$ with an almost optimal work of $O(mn^{2.5}\log n)$.

Following this, we show how to reduce the normalized edit distance problem to multiple matrix multiplications. The resulting parallel algorithm has a time complexity of $O(\log^2 n)$, and uses $O(n^6/\log n)$ processors connected as a hypercube.

2. The Dynamic Programming Solution

We first define the editing process more formally and review some of the definitions and results of Marzal and Vidal [8]. Suppose $X = (X_1X_2\cdots X_n)$ and $Y = (Y_1Y_2\cdots Y_m)$ are two strings of lengths n and m over the finite alphabet $A = \{a_1, \ldots, a_k\}$. We associate two pointers p_X and p_Y to a character within X and Y, initially positioned to the first character in X or Y, i.e. $p_X = p_Y = 1$. The cost for substituting $a_j \in A$ for the character $a_i \in A$ is given by $\gamma(a_i \to a_j)$, the cost for inserting character $a_i \in A$ (i.e. replacing ε by a_i) is $\gamma(\varepsilon \to a_i)$, and the cost for deleting $a_i \in A$ (i.e. replacing a_i by ε) is $\gamma(a_i \to \varepsilon)$. Additionally, we use a string Z which is initially equal to X and finally equal to Y, and a pointer p to an character within Z. In each transformation step of the editing path, we consider one of the following choices:

1. Replace the current character $Z_p = X_{p_X}$ in Z with the current character Y_{p_Y} in Y. Then, increment p, p_X and p_Y. The cost for this operation is $\gamma(X_{p_X} \to Y_{p_Y})$.
2. Delete the current character $Z_p = X_{p_X}$ in Z and increment p_X. The cost incurred is $\gamma(X_{p_X} \to \varepsilon)$. Note that although p does not change, Z_p now is the character following the deleted one.
3. Insert the current character in $Z_p = Y_{p_Y}$ in Z and increment both p and p_Y. The cost incurred is $\gamma(\varepsilon \to Y_{p_Y})$.

Let $D(i,j)$ denote the edit distance between the prefixes $X_1X_2\cdots X_i$ and $Y_1Y_2\cdots Y_j$ for $i,j > 0$. The above property of an optimal editing path leads immediately to the following well known dynamic programming formulation:

$$\begin{aligned} D(i,j) \quad = \min\{ \quad & D(i-1,j) + \gamma(X_i \to \varepsilon), \\ & D(i,j-1) + \gamma(\varepsilon \to Y_j) \\ & D(i-1,j-1) + \gamma(X_i \to Y_j)\}, \end{aligned} \tag{1}$$

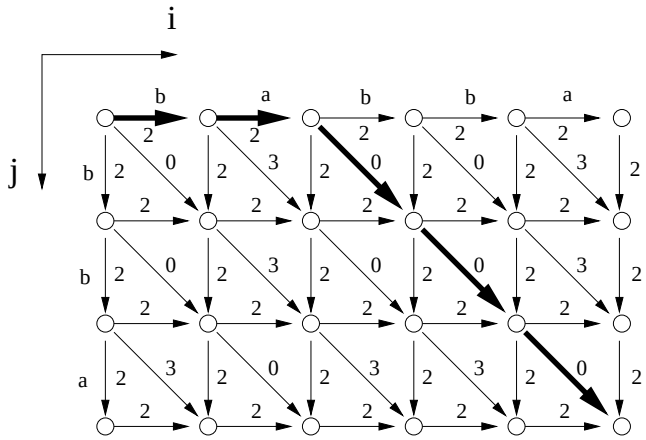
for $i \geq 0$ and $j \geq 0$, with boundary conditions $D(i,-1) = D(-1,j) = \infty$ and $D(0,0) = 0$. The edit distance between X and Y is then equal to $D(n,m)$. Put another way, the problem of finding a minimal weight editing path between X and Y is equivalent to finding a shortest path from $(0,0)$ to (n,m) in the directed weighted *transformation graph* $G_{n,m}$ with vertex set $V = \{(i,j) \mid 0 \leq i \leq n, 0 \leq j \leq m)\}$, the edge set $E = \{((i,j),(i+1,j)) \mid 0 \leq i < n\} \cup \{((i,j),(i,j+1)) \mid 0 \leq j < m\} \cup \{((i,j),(i+1,j+1)) \mid 0 \leq i < n,\ i \leq j < m\}$, and the weights $W((i,j),(i+1,j)) = \gamma(X_i \to \varepsilon)$, $W((i,j),(i,j+1)) = \gamma(\varepsilon \to Y_j)$, and $W((i,j),(i+1,j+1)) = \gamma(X_i \to Y_j)$. An example for a $G_{5,3}$ is given in Figure 1.

To compute the normalized edit distance between X and Y, one can proceed by finding a shortest path in $G_{n,m}$ from $(0,0)$ to (n,m) for each possible path length k, then divide the weight of the path found by k, and finally choose the smallest ratio. Since we are assuming $n \geq m$, k lies in the range $n \leq k \leq n+m$. In order to keep track of the path lengths in the computation, we augment the state space of the recurrence relations in (1) and obtain

$$\begin{aligned} D(i,j,k) \quad = \quad & \min\{D(i-1,j,k-1) + \gamma(X_i \to \varepsilon), \\ & D(i,j-1,k-1) + \gamma(\varepsilon \to Y_j) \\ & D(i-1,j-1,k-1) + \gamma(X_i \to Y_j)\}, \end{aligned} \tag{2}$$

for $i,j > 0$, $\max(i,j) \leq k \leq i+j$, with boundary conditions $D(i,j,k) = \infty$ if $i = -1$ or $j = -1$ or

[1] In [13], Marzal, Vidal and Aibar showed how to use fractional programming techniques to compute the normalized edit distance by repeatedly computing ordinary edit distances with $O(mn)$ cost per iteration. However, to the best of our knowledge, there is no good bound limiting the maximum number of iterations to n.



γ	ε	a	b
ε	—	2	2
a	2	0	3
b	2	3	0

Figure 1. The Transformation Graph $G_{5,3}$ for the strings $X = babba$ and $Y = bba$.

$k < \max(i, j)$, and $D(0, 0, 0) = 0$. Here, $D(i, j, k)$ denotes the distance between the prefixes $X_1X_2 \cdots X_i$ and $Y_1Y_2 \cdots Y_j$, using only edit paths of length k, for $0 \leq i \leq n$, $0 \leq j \leq m$, $0 \leq k \leq n + m$. Since all possible edit paths obtainable by the three edit operations have lengths between n and $n+m$, the normalized edit distance can then be found by computing

$$\min\{D(n, m, k)/k \mid n \leq k \leq m + n\}. \qquad (3)$$

The proof of correctness of this computation for the normalized edit distance can be found in [8].

We now generalize the weighted graph and shortest path formulation for the edit distance to the normalized edit distance. In terms of shortest paths, the normalized edit distance can be computed by finding the shortest paths from $(0, 0, 0)$ to (n, m, k), $n \leq k \leq m + n$ in the weighted directed transformation graph $G'_{n,m}$ which is constructed from $G_{n,m}$ by creating a cube consisting of $n + m + 1$ copies of the vertex set of $G_{n,m}$, one for each value of $k = 0, 1, \ldots, n + m$ that arises in the recursion (2). More precisely, the vertex set of $G'_{n,m}$ is $V' = \{(i, j, k) \mid 0 \leq i \leq n,\ 0 \leq j \leq m,\ 0 \leq k \leq n + m\}$. According to the dynamic programming formulation (2), there are three types of edges: $((i, j, k), (i+1, j, k+1))$ with weight $\gamma(X_i \to \varepsilon)$, $((i, j, k), (i, j + 1, k + 1))$ with weight $\gamma(\varepsilon \to Y_j)$, and $((i, j, k), (i + 1, j + 1, k + 1))$ with weight $\gamma(X_i \to Y_j)$. After the computation of the cost of a shortest path from $(0, 0, 0)$ to (n, m, k) for every k, we make use of (3) to compute the normalized edit distance between X and Y.

3. Algorithms for the Mesh and Linear Array

Simple parallel algorithms for the computation of the normalized edit distance on a two dimensional mesh or a linear array of processors rely on the structure of the dynamic programming formulation of the solution. The most evident way to solve the recurrence relation (2) is to calculate the shortest path length (with respect to the weight function) from $(0, 0, 0)$ to all nodes with distance (with respect to the number of edges traversed) k, for k ranging from 1 to $n + m$.

The sequential running time does not increase if we compute the values $D(i, j, k)$ layer by layer in the transformation graph $G'_{n,m}$, that is, first we compute the quantities $D(i, j, 1)$ for all i, j, then we compute the quantities for $D(i, j, 2)$, and so on. This motivates the following straightforward parallelization. Suppose we employ an $n \times m$ rectangular mesh of processors (more precisely we use a $(n+1) \times (m+1)$ mesh and assign processors to boundary nodes as well). Then we assign each node (i, j, k) of $G'_{n,m}$ and the computation of the values $D(i, j, k)$ onto processor (i, j). Since layer 0 contains only initial values, we start calculating the $D(\cdot, \cdot, \cdot)$-values in layer $k = 1$, and proceed through the values $k = 2, 3, \ldots, n+m$. Each value in level k needs at most three values from level $k - 1$, two from neighboring processors assigned to $(i - 1, j)$ and $(i, j - 1)$, and one from the processor at node $(i - 1, j - 1)$, which is two hops away from (i, j). Therefore the calculation of all required values in a layer needs $O(1)$ time and memory.

Thus, the overall time complexity of the algorithm is $O(m + n)$ with an asymptotically optimal $O((m + n)mn)$ work. Note that the work is only asymptotical optimal, since in layer $k = 2$, for example, only 3 out of nm nodes are computing while the other processors remain idle.

If as many as nm processors are not available, it is possible to split up the calculation of each layer into n columns or m rows and calculate the values in each column/row in parallel. This leads to a parallel algorithm for an array of n or m processors, with the same asymptotic work $O((m + n)mn)$, and a time complexity of $O(nm)$ on an n processor array, and $O(n^2)$ on an m processor array.

```
(01) for k from 1 to m+n do
(02)    for i from 0 to n do in parallel
(03)       for j from 0 to m do in parallel
(04)          D(i,j,k) = min{D(i-1,j,k-1)+γ(X_i → ε),
(05)                        D(i,j-1,k-1) + γ(ε → Y_j),
(06)                        D(i-1,j-1,k-1)+γ(X_i → Y_j)}
(07)       endfor
(08)    endfor
(09) endfor
(10) find-minimum of D(n,m,k)/k for n ≤ k ≤ m+n.
```

Figure 2. Straightforward parallelization of the $D(i,j,k)$ computation.

4. A Sublinear Time Hybrid Algorithm

When reconsidering the work-optimal algorithm for a mesh of $n \times m$ processors, we see that the layers in the transformation graph $G'_{n,m}$ are computed one after each other, and each new layer uses only values from the previous layer. In what follows, we no longer use an array of processors but assume a CREW PRAM model of parallel computation with nml processors, for $1 \leq l \leq n$, to be determined later.

Let L_k denote the k-th layer of $G_{n,m}$, $1 \leq k \leq n+m+1$. Thus for a given k, L_k is the subgraph of $G'_{n,m}$ induced by the nodes $\{(i,j,k) \mid 0 \leq i \leq n,\ 0 \leq j \leq m\}$. We then divide $G'_{n,m}$ into l stripes $S_1, S_2, \ldots, S_l$, where each stripe consists of $(m+n+1)/l = h$ layers. Thus, stripe S_1 consists of layers $L_1, L_2, \ldots, L_h$, stripe S_2 consists of layers $L_{h+1}, L_{h+2}, \ldots, L_{2h}$, and so on. This division is illustrated in Figure 3.

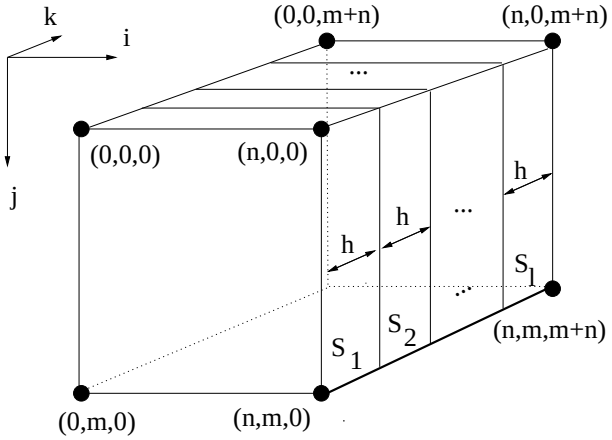


Figure 3. Dividing the transformation graph $G'_{n,m}$ into stripes.

For each stripe S_k in parallel, we calculate the distance (cost of a shortest path) from every node $(i,j,(k-1)h+1)$ in the first layer of the stripe to each node (i',j',kh) in the last layer of S_k. Due to the direction of the arcs on $G'_{n,m}$, this distance is ∞ if $i' < i$, or $j' < j$. Furthermore, going from one level to the next in $G'_{n,m}$ in the worst case is a diagonal move, which increments both i and j coordinates by 1. Thus we have an upper bound $i' \leq i+h$, and $j' \leq j+h$. This means that for each node (i',j',kh) in the last layer of the stripe S_k, the distance from the first layer of S_k is finite only for a small subset of nodes, namely the vertices $(i,j,(k-1)h+1)$ satisfying $i'-h \leq i \leq i'$ and $j'-h \leq j \leq j'$. Set $window(i',j') = \{(i,j,(k-1)h+1) \mid i'-h \leq i \leq i,\ j'-h \leq j \leq j'\}$. Since $window(i',j')$ has $O(h^2)$ nodes, the distance between (i',j',kh) and each candidate node in its window can be computed in $O(h^3)$ sequential time. To see this, note that these distances can be computed layer by layer in a reverse fashion, starting from (i',j',kh) and proceeding to the nodes in $window(i',j')$ through layers $L_{kh-1}, L_{kh-2}, \ldots, L_{(k-1)h+1}$. The computation per node takes constant time and the number of nodes involved is $O(1+2^2+3^2+\cdots+h^2) = O(h^3)$. In particular, the distance from a given node (i',j',kh) in the last layer of S_k to all the nodes in $window(i',j')$ in the first layer of S_k can be computed in $O(h^3)$ sequential time for each stripe S_k.

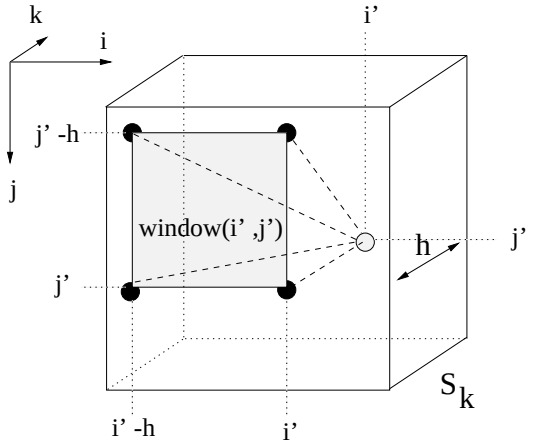


Figure 4. The window needed for the perimeter pair distance in stripe S_k.

After having computed these distances for every stripe S_k, we can iteratively compute the distance from $(0,0,0)$ to nodes of the form (i',j',kh) by finding

$$\min_{(i,j,(k-1)h+1)} \{dist_{G'_{n,m}}((0,0,0),(i,j,(k-1)h+1)) + dist_{G'_{n,m}}((i,j,(k-1)h+1),(i',j',kh))\}, \quad (4)$$

where $(i,j,(k-1)h+1)$ ranges in $window(i',j')$. Determining this minimum takes $O(h^2)$ sequential time, and by using the l processors at our disposal for each node

(i', j') in layer L_{kh}, the parallel time to compute the minimum becomes $h^2/l + \log l$. Note that the computation of $dist_{G'_{n,m}}((0,0,0),(i,j,(k-1)h+1))$ involves $O(1)$ computation of the minimum distances between the node $(i,j,(k-1)h+1)$ and its 3 neighbors in level $(k-1)h$. Alternatively, we could have chosen the stripes overlapping and have the distance $dist_{G'_{n,m}}((0,0,0),(i,j,(k-1)h+1))$ already precomputed. However, this additional step does not improve the complexity of the algorithm.

```
(01) for each stripe S_k do in parallel
(02)    for i from 0 to n do in parallel
(03)       for j from 0 to m do in parallel
(04)          compute dist((i,j,(k-1)h+1),(i',j',kh))
(05)             for each (i,j,(k-1)h+1)
(06)                in window(i',j').
(07)       endfor
(08)    endfor
(09) endfor
(10) for k from 1 to l do
(11)    for i from 0 to n do in parallel
(12)       for j from 0 to m do in parallel
(13)          compute D(i,j,kh)
(14)       endfor
(15)    endfor
(16) endfor
(17) compute D(i,j,k), n ≤ k ≤ m+n,
(18)    repeating steps 1-9 using the updated
(19)    start values in each stripe.
(20) find-minimum of D(n,m,k)/k for n ≤ k ≤ m+n.
```

Figure 5. Sublinear time hybrid computation of the $D(i,j,k)$.

Now the total time needed for computing the distances between the first and last layer of a single stripe is $O(h^3)$, and all stripes are processed in parallel. The time for iteratively putting the stripes together according to (5) is given by $l \cdot (h^2/l + \log l)$. This computation includes keeping track of the node in the previous layer that is actually on a shortest path from $(0,0,0)$ to (n,m,kh), $1 \le k \le l$. This can be done by storing for every node of the form (i', j', kh), the nodes on the path to the element in $window(i', j')$ for which (5) is minimized. Thus, the total time required for the computation of all values of $D(i,j,hk)$ is $O(h^3 + l \log l)$. Finally, the values of $D(i,j,k)$, where k is not a multiple of h can be computed using the window technique and the precomputed values at the end of the current layer, using the same computation as above, in $O(h^3)$ time without increasing the time complexity. The total work is $O(mnl(h^3 + l \log l))$.

Since $l \cdot h = O(n)$, if we take $h = n^\alpha$, we obtain the time complexity $O(n^{3\alpha} + n^{1-\alpha} \log n)$ and a work of $O(mn^{1+3\alpha} + mn^{3-2\alpha} \log n)$. For small α, the term $n^{1-\alpha} \log n$ dominates, whereas $n^{3\alpha}$ is larger for large values of α. The optimal time complexity is obtained by choosing $\alpha = 1/4$, yielding a time complexity of $O(n^{0.75} \log n)$ and a work of $O(mn^{2.5} \log n)$. The number of processors used is $mnl = O(mn^{1.75})$. We summarize the result in the following theorem:

Theorem 1 *On a $O(mn^{1.75})$ processor CREW PRAM, the normalized edit distance problem can be solved with a time complexity of $O(n^{0.75} \log n)$ and total work $O(mn^{2.5} \log n)$.*

5. An NC Algorithm Using Matrix Multiplication

To obtain a polylogarithmic time complexity, we reduce the normalized edit distance problem to a problem involving several matrix multiplications. It is well known that two $n \times n$ matrices can be multiplied in $O(\log n)$ time, using $O(n^3/\log n)$ processors. Of course, since this matrix multiplication is not work-optimal, our algorithm will not be work-optimal either. In fact, it turns out that the hypercube algorithm we propose has a work complexity of $O(n^6 \log^2 n)$, which is far off from the optimal $O(n^3)$.

The main idea of this algorithm is similar to the one employed by Ibarra, Pong, and Sohn in [4] for the conventional edit distance. We split the edit path graph $G'_{n,m}$ into small pieces, compute the distances in each piece, and consequently merge these pieces by computing the distance between any pair of nodes on the facing perimeter of each piece. More precisely, we start with nm disjoint subgraphs of $G'_{n,m}$ of size $m+n+1$ each, consisting of the nodes (i,j,k), $0 \le k \le n+m$. For simplicity of presentation, assume that $m = n = 2^d - 1$ for some integer d. Then, alternating in i and j-direction, we group two adjacent blocks and merge them. This process is illustrated in Figure 6. The third dimension has been omitted to clarify the presentation. Each node in the figure actually represents $n+m+1$ nodes laid out as an array that extends into the page.

During the whole merging process, we maintain the following information: for each pair of nodes (i,j,k) and (i',j',k') on the perimeter of the current sub-block of $G'_{n,m}$, we keep track of the distance between those nodes. Assume we are merging two blocks A and B set up as in Figure 7. In Figure 7 the right block B can be split up in the leftmost $(1 \times m \times (n+m+1))$-sub-block I, and the remaining subset of nodes, B'. For each pair of nodes $u = (i,j,k)$ and $v = (i',j',k')$ in $A \cup B$ for which there is a path from u to v in $G'_{n,m}$, we have the following possibilities:

1. $u \in A$ and $v \in A$. In this case, the distance between u and v is already known.

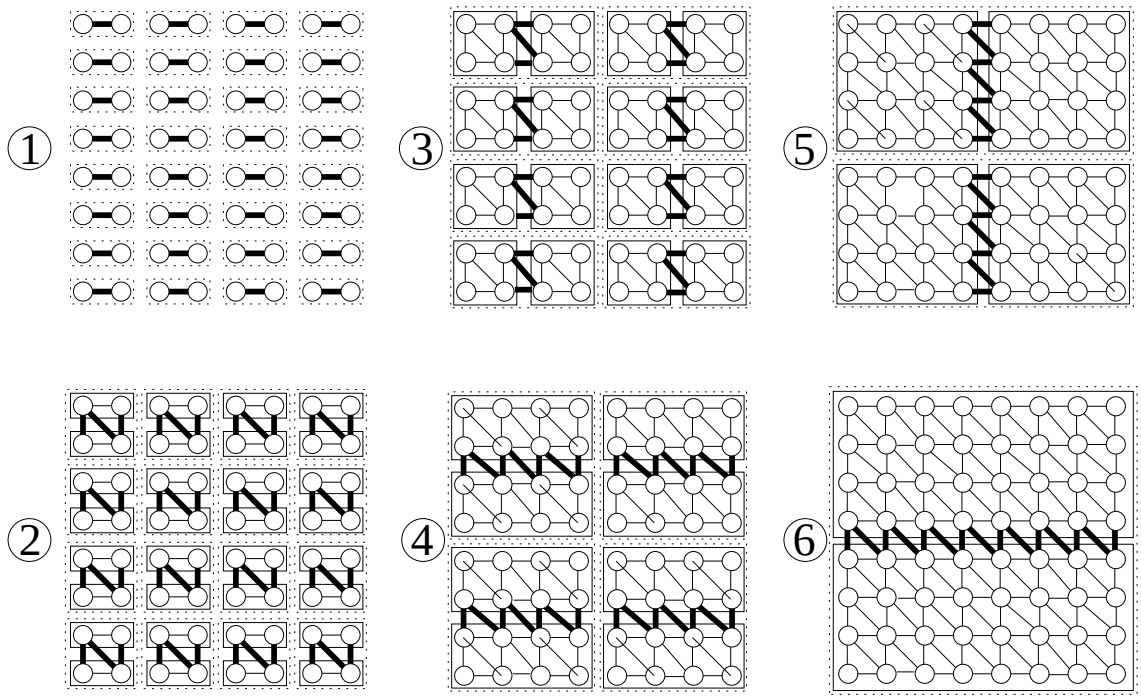


Figure 6. Iterated merging of sub-blocks along alternating dimensions.

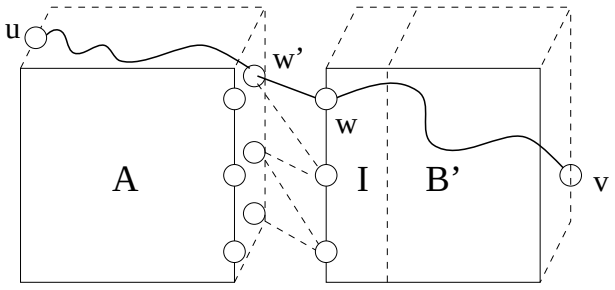


Figure 7. Perimeter pair distance computation

2. $u \in B$ and $v \in B$. Again, the distance between u and v is already known.

3. $u \in A$ and $v \in B$. Due to the nature of the edges of the transformation graph $G'_{n,m}$, it is clear that any path from u to v must pass through a node w in I. Furthermore $dist_{G'}(u,v) = dist_{G'}(u,w) + dist_{G'}(w,v)$. Note that $dist_{G'}(w,v)$ is already known since both v and w are in B, and that $dist_{G'}(u,w)$ can be computed in constant time by calculating $dist_{G'}(u,w') + W(w',w)$ for at most 3 neighbors w' of w that are located on the perimeter of A.

In summary, we have $2d$ merging steps, and in each step, we have to calculate

$$dist(u,v) = \min_{w \in I} \ (dist(u,w) + dist(w,v)),$$

which is equivalent to a matrix multiplication (to see this, replace min by $\sum$ and $+$ by $*$) of a $K \times K'$ matrix with a $K' \times K$ matrix, where K is the number of nodes on the perimeter of a current block, and K' is the number of nodes in the leftmost or sub-block of the current block, which corresponds to the number of nodes in the sub-block I of Figure 7. Since K and K' are bounded by $4(2n+1)2^i$ for the $2i$-th and the $(2i+1)$-th merging steps, the matrices to be multiplied are $O(n^2 \times n^2)$. Therefore the time needed to complete a step is $O(\log(n^2)) = O(\log n)$, and the number of processor used in the i-th step per block is $O(n^3 2^{3i}/(i+\log n))$. If we want to merge all $n^2/2^{2i}$ sub-blocks in parallel, we need $O(n^5 2^i/(i+\log n))$ processors for the i-th step, so that $O(n^6/\log n)$ processors are always enough. This computation outlined in Figure 5, can be carried out on a hypercube connected multicomputer as shown in [4]. Furthermore if α is the best sequential matrix multiplication exponent, then the product of two $n \times n$ matrices can be computed in $\log n$ time with a total of $O(n^\alpha)$ operations on a CREW PRAM [5]. Thus, the algorithm in Figure 5 that computes the normalized edit distance can be implemented to run in $O(\log^2 n)$ time using a total of $O(n^{2\alpha} \log n)$ operations on a $n^{2\alpha}/\log n$ processor CREW PRAM.

```
(01) for k from 1 to log n do
(02)     for j from 0 to m do in parallel
(03)       merge two blocks each of size 2^k × 2^k
(04)          to one block of size 2^k × 2^(k+1)
(05)           horizontally
(06)       merge two blocks each of size 2^k × 2^(k+1)
(07)          to one block of size 2^(k+1) × 2^(k+1)
(08)           vertically
(09)    endfor
(10) endfor
```

Figure 8. Alternate merges.

6. Summary and Future Work

We have developed diverse parallel algorithms for computing the normalized edit distance, a string comparison technique superior to the conventional edit distance in empirical experiments. We presented work optimal algorithms with n, m and nm processors and a sublinear time algorithm with a time complexity of $O(n^{0.75} \log n)$ time and $O(mn^{2.5} \log n)$ work. Finally, we presented a polylogarithmic algorithm with a time complexity of $O(\log^2 n)$ and a work of $O(n^6 / \log n)$ on a hypercube connected multicomputer, or $O(n^{2\alpha} / \log n)$ on a CREW PRAM, where α is the best sequential matrix multiplication exponent. Table 1 summarizes the results for the different algorithms and architectures.

We remark that it is possible combine the two sublinear time algorithms proposed here. The first sublinear time algorithm we presented splits the sequential problem into several stripes and uses the dynamic programming technique to solve the problem on the stripes individually. Instead, we could use the algorithm based on matrix-multiplication to solve the sub-problems on the stripes to derive a faster running time. This algorithm would have a smaller time complexity than $n^{0.75} \log n$, and at the same time use less than $n^6 / \log n$ processors. We omit the detailed description of this technique in this extended abstract. Note that for the conventional edit distance, our first sublinear time algorithm can be modified in a straightforward manner to have a time complexity of $n^{2/3} \log n$ using $n^{5/3}$ processors.

Since even the optimal work of the normalized edit distance solution by dynamic programming is by a factor of n larger than the work for the conventional edit distance, the question arises as to whether there is better algorithm for the computation of the normalized edit distance problem than dynamic programming. Vidal, Marzal and Aibar showed in [13] that the normalized edit distance can be computed by the Dinkelbach algorithm, which is a variant of an optimization technique for *fractional programming* [12, 3]. Using fractional programming, the normalized edit distance is computed by using several iterations of the conventional edit distance with time dependent weights. However, although in the performed experiments the expected number of iterations was empirically found to be a small constant, there appears to be no useful upper bound or provable average behavior of the number of iterations.

Dynamic Programming	**# of procs**	**Time**	**Total work**
Array $L(m)$	m	n^2	mn^2
Array $L(n)$	n	mn	mn^2
Mesh $M(m,n)$	mn	n	mn^2
Hybrid Algorithm	**# of procs**	**Time**	**Total work**
CREW PRAM	$mn^{1.75}$	$n^{0.75} \log n$	$mn^{2.5} \log n$
Matrix Multiplication	**# of procs**	**Time**	**Total work**
Hypercube	$n^6 / \log n$	$\log^2 n$	$n^6 \log n$
CREW PRAM	$n^{2\alpha} / \log n$	$\log^2 n$	$n^{2\alpha} \log n$

Table 1. Summary of the characteristics of the proposed parallel algorithms.

Thus, for future research, we plan to derive better bounds for the fractional programming approach, and then use the existing parallel methods for the conventional edit distance to devise a faster parallel algorithm for the normalized edit distance. It may also be useful to investigate generalizations of normalized edit distance in which context-sensitive costs and character transpositions of the type that arise in computational biology applications are allowed. Furthermore, the study of the average time behavior of the Dinkelbach algorithm as it is applied to the normalized edit distance as well as other combinatorial optimization problems is an interesting problem in its own right.

References

[1] E. Edminston and R. A. Wagner. Parallelization of the dynamic programming algorithm for comparison of sequences. In *Proceedings of the 1987 International Conference of Parallel Processing*, pages 78–80, August 1987.

[2] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal Of Complexity*, 4:33–72, 1988.

[3] T. Ibaraki. Parametric approaches to fractional programs. *Mathematical Programming*, 26:345–362, 1983.

[4] O. Ibarra, T.-C. Pong, and S. M. Sohn. Hypercube algorithms for some string comparison problems. In *Proceedings of the 1988 International Conference of Parallel Processing*, pages 190–193, August 1988.

[5] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. p. 248.

[6] E. Lander. Protein sequence comparison on a data parallel computer. In *Proceedings of the 1988 International Conference of Parallel Processing*, pages 257–263, August 1988.

[7] L. L. Larmore and W. Rytter. An optimal sublinear time parallel algorithm for some dynamic programming problems. *Information Processing Letters*, 52:31–34, 1994.

[8] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):926–932, September 1993.

[9] W. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980.

[10] S. Ranka and S. Sahni. String editing on an SIMD hypercube multicomputer. *Journal of Parallel and Distributed Computing*, 9(4):411–418, 1990.

[11] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparisons*. Addison-Wesley, Reading, MA, 1983.

[12] M. Sniedovich. *Dynamic Programming*, pages 348–354. Marcel Dekker Inc., 1992.

[13] E. Vidal, A. Marzal, and P. Aibar. Fast computation of normalized edit distances. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(9):899–902, September 1995.

[14] R. Wagner and M. Fisher. The string-to-string correction problem. *J. Assoc. Computing Machinery*, 21(1):168–173, 1974.