1989

Sequence Comparison of The Connection Machine

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Scott McFaddin

Report Number:
89-853

SEQUENCE COMPARISON OF THE CONNECTION MACHINE

Mikhail J. Atallah
Scott McFaddin

CSD-TR-853
February 1989
(Revised September 1990)

Sequence Comparison on the Connection Machine

Mikhail J. Atallah * Scott McFaddin †

Abstract

We give two parallel algorithms for sequence comparison on the Connection Machine 2 (CM-2). The specific comparison measure we compute is the *edit distance*: given a finite alphabet Σ and two input sequences $X \in \Sigma^+$ and $Y \in \Sigma^+$ the edit distance $d(X, Y)$ is the minimum cost of transforming X into Y via a series of weighted insertions, deletions, and substitutions of characters. The edit distance comparison measure is equivalent to or subsumes a broad range of well known sequence comparison measures.

The CM-2 is very fast at performing **parallel prefix** operations. Our contribution consists of casting the problem in terms of these operations. Our first algorithm computes $d(X, Y)$ using N processors and $O(MS)$ time units, where $M = \min(|X|, |Y|) + 1$, $N = \max(|X|, |Y|) + 1$ and S is the time required for a parallel prefix operation. The second algorithm computes $d(X, Y)$ using NM processors and $O((\log N \log M)(S + \mathcal{R}))$ time units, where $\mathcal{R}$ is the time for a "router" communication step – one in which each processor is able to read data, in parallel, from the memory of any other processor. Our algorithms can also be applied to several variants of the problem, such as subsequence comparisons, and one-many and many-many comparisons on "sequence databases".

Key words and phrases: Connection Machine, parallel computation, sequence comparison, weighted edit distance, shortest paths, grid graphs.

*Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. This author's research was supported by the Office of Naval Research under Contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under Grant AFOSR-90-0107, the National Science Foundation under Grant DCR-8451393 with matching funds from AT&T, and the National Library of Medicine under Grant R01-LM05118. Part of this research was carried out while this author was visiting the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California.

†Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. This author's research was supported by The Research Institute for Advanced Computer Science, Center for Advanced Algorithms and Architectures, NASA Ames Research Center, California, and by the Office of Naval Research under Contract N00014-86-K-0689.

1 Introduction

The need for measuring similarities between sequences of characters arises in many areas of it. Applications of the sequence comparison problem include molecular biology, robot vision, cryptography, speech recognition, and surface reconstruction of medical scans. (See [6, 12, 15, 19, 20] for example.) The common theme in these applications is the notion of transforming a source sequence $X = x_1x_2...x_{|X|}$ of characters from some finite alphabet Σ into a destination sequence $Y = y_1y_2...y_{|Y|}$ of characters from Σ, by using a series of transformation steps consisting of deletions, insertions, and substitutions of characters.

We assume there is a system of non-negative weights associated with these transformations steps, as follows: For each character $c \in \Sigma$ there is a weight $I(c)$ charged for inserting c into X, a weight $D(c)$ charged for deleting c from X, and for each pair $(c, d) \in \Sigma \times \Sigma$ there is a weight $S(c, d)$ charged for substituting a c for a d.

1.1 The edit distance problem

Although there are many comparison measures available, we shall assume we are computing the *weighted edit distance* $d(X, Y)$: the minimum total cost of transforming X into Y by applying a series of the weighted transformation steps above. Without loss of generality we may assume $|Y| \leq |X|$ (otherwise interchange the roles of insertion and deletion and transpose the substitution table S). The edit distance subsumes or is equivalent to many well known comparison measures.

The transformation steps and the arrangements of the weights reflect the nature of the application domain. The comparison of two DNA sequences, for example, may yield a measure of evolutionary ancestry or of common function in two genetic codes: deletion or insertion transformation steps correspond to loss or acquisition of genetic material, and substitutions correspond to pointwise genetic mutation. Other applications, as in optimal surface reconstruction [6] may ignore the possibility of substitution altogether – by setting $S(\cdot, \cdot) = \infty$ (this does not reduce the complexity of the problem, though). The alphabet Σ is typically of small size – the set {a,c,g,t} of DNA bases, for example. The sequence lengths, however, may reach many thousands or millions of characters.

1.2 A graphical formulation

The problem has a well known formulation in terms of shortest path computation on a certain class of directed graphs, which we review next.

> If $T = t_1t_2...t_n$ is a string, then by "the k-prefix of T" we mean the string consisting of the first k characters of T, i.e. $t_1t_2...t_k$. A zero-prefix is the empty string.

From now on $M = |Y| + 1$, and $N = |X| + 1$ (hence $M \leq N$). Let $C(i,j)$ be the weighted edit distance from the j-prefix of X to the i-prefix of Y ($0 \leq i \leq |X|$, $0 \leq j \leq |Y|$). We set $C(0,0) = 0$. The answer we wish to compute is $C(M-1, N-1)$.

It has been shown [21], [15] that

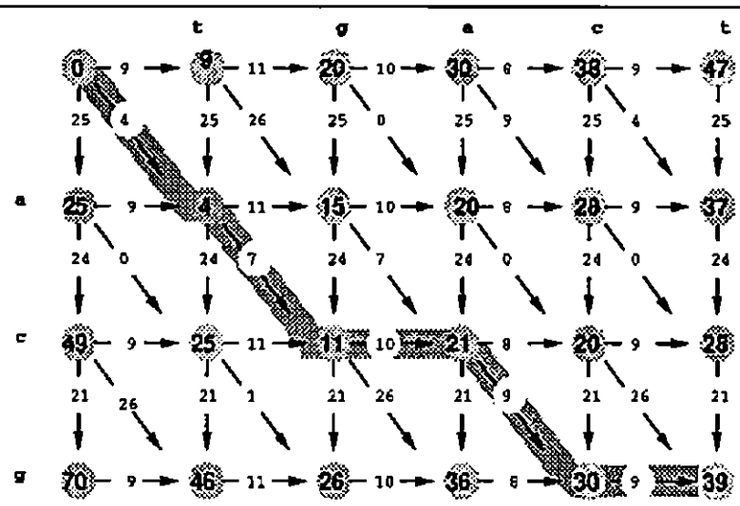
$$C(i,j) = \min\{C(i-1,j) + I(y_i), C(i,j-1) + D(x_j), C(i-1,j-1) + S(y_i, x_j)\}.$$

This means that $C(i,j)$ may be computed whenever $C(i-1,j), C(i,j-1)$, and $C(i-1,j-1)$ have been computed. This dependency is represented by a weighted digraph, called a ***grid graph***, whose vertices are laid out in an M by N grid (See Figure 1.2).

The cost $C(i,j)$ is associated with grid graph vertex (i,j). Vertex (i,j) has directed edges leading to vertices $(i+1,j)$, $(i,j+1)$, and $(i+1,j+1)$. These edges are given weights $I(y_{i+1})$, $D(x_{j+1})$, and $S(y_{j+1}, x_{i+1})$ respectively. The solution to the edit distance problem, then, is to find the cost of a shortest path in the weighted grid graph from vertex $(0,0)$ to vertex $(M-1, N-1)$.

1.3 Previous Work

There have been many parallel algorithms given which find shortest path lengths in grid graphs, and thus apply to sequence comparison. A number of papers [1, 2, 13] have dealt with the most general parallel machine model, the Parallel Random Access Memory machine (PRAM) [16] in hopes of exposing the maximum parallelism inherent in the problem itself. For hypercubes, Ibarra, Pong, and Sohn [9] have given a synchronous algorithm for computing the edit distance requiring $O(N^3/\log^2 N)$ processors and $O(\log^2 N)$ time steps (assuming $M = N$). Ranka and Sahni [17] have also given a synchronous hypercube algorithm that has a time/processor tradeoff – one bound it implies is and $O(\sqrt{N \log N})$ time steps with $O(N^2)$ processors. Edmiston and Wagner [5] give an algorithm for a synchronous linear array of processors requiring $O(N+M)$ time and M processors. This applies to the CM-2. Lander, Mesirov, and Taylor [11] assume a "data parallel computer", such as the CM-2, and take a similar algorithmic approach, but for comparisons among strings in a sequence database. The time requirements for their algorithm are complex, and depend upon the distribution of lengths of individual sequences in the database.



Weight Table

	I(·)	D(·)	S(·,a)	S(·,c)	S(·,g)	S(·,t)
a	25	10	0	9	26	4
c	24	8	7	0	7	0
g	21	11	26	9	1	26
t	21	9	6	0	26	1

Figure 1.2: A Weighted Grid Graph. The alphabet is { a,c,g,t }. The graph is induced by comparing the sequences tgact and acg. The values of $C(i,j)$ appear on the nodes. The output of the algorithm is the length of the shortest path (shaded) – an edit distance of 39. This corresponds to the sequence alignment

```
t g a c t
a c   g
```

which is optimal. It would appear that the sequences should be aligned at the end, since the a and the c would match. However, this would require a substitution of a g for a t, which is very expensive. In terms of DNA sequencing, this would mean that the user assumes thymine (t) is rarely converted into guanine (g) by a point-to-point mutation.

1.4 Our Contribution

We give two CM-2 algorithms for the sequence comparison problem. These algorithms differ from those above in that they are oriented towards specially optimized programming primitives built into the CM-2, namely those which perform parallel prefix computations. These are called "scan" operations in CM-2 lingo. Our first algorithm runs in $O(MS)$ time using N processors and the second operates in $O((\log N \log M)(\mathcal{S} + \mathcal{R}))$ time using NM processors, where $\mathcal{S}$ is the time for a "scan" operation and $\mathcal{R}$ is the time for a global communication. The first algorithm is useful when sequence lengths are much larger than the processor count of the CM-2 – quite likely with current CM-2 configurations. The second algorithm utilizes more processors and is thus "more parallel". This algorithm is more useful with CM-2 configurations having far more processors than present versions, when processor utilization becomes more important. The algorithms have been implemented and performance results are given.

1.5 Realistic Computations

To get an idea of the nature of sequences that are found in real applications, we have inspected the GenBank [3] data bank (Release 64.1), a large set of genetic sequences used in microbiology. This data bank contains 35,100 sequences having a total of 42,495,893 characters, for an average sequence length of about 1,200 (only 18 sequences in GenBank are over 30,000 characters long). GenBank is divided into several sub-banks – PLANT, VIRAL, STRUCTURAL RNA, SYNTHETIC, etc. The VIRAL sub-bank, for example, has 3,216 sequences of average length around 1,700. Most sequences in the VIRAL sub-bank range between 200 and 5,000 characters in length.

In such applications, the user often wishes to compare one or more source sequences $X_1, X_2, ..., X_S$ to each sequence in some database of destination sequences $Y_1, Y_2, ..., Y_D$. Let $G_{K,L}$ denote the grid graph induced when comparing the source string X_L to destination string Y_K. Then we may solve the composite problem by embedding the $G_{K,L}$ graphs into a larger graph as shown in Figure 1.5.

The pairwise edit distances are found at the bottom right corners of the embedded subgraphs. Lander, et. al. [11] have given algorithms for database style sequence comparisons for the case of symmetric weight tables. We point out the importance of the special case when $D = 1$ and $|Y_1|$ is small, but the size of the opposing database $\sum_{i=1}^{S} |X_i|$ is quite large. This special case is frequently encountered in molecular biology when comparing a single test sequence to a large database of known sequences. The algorithmic implication is that the grid graph becomes very long and narrow – for example, comparing a single VIRAL sequence in GenBank to the entire VIRAL sub-bank would yield, roughly, a 1-thousand by 5-million grid graph. This type of geometry has substantial ramifications for CM-2 implementation (more on this later).

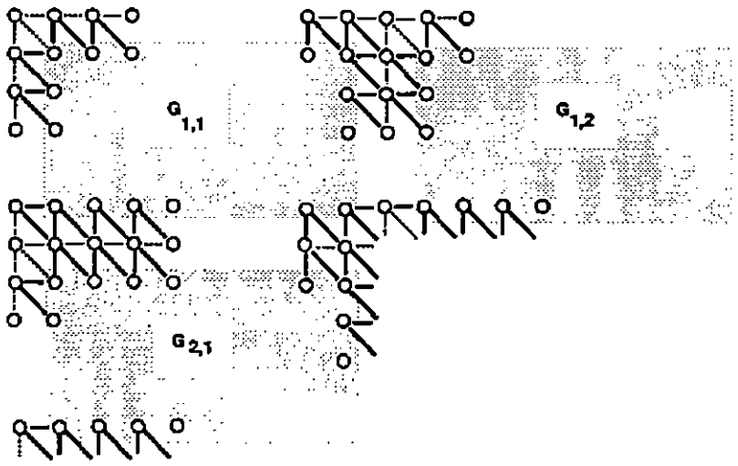


Figure 1.5 Embedding a Database Comparison in a Grid Graph. The heavy edges have infinite weight and the thin edges have zero weight. This isolates embedded graphs from one another, but allows the parallel algorithm to operate without regard to their arrangement.

1.6 Outline of the Paper

The remainder of the paper is laid out as follows:

Section 2: The Connection Machine architecture and software.

Section 3: The first algorithm and its performance results.

Section 4: The second algorithm and its performance results.

Section 5: Summary and Future Work.

2 The CM-2 Architecture and Software

The Connection Machine 2 [8] is a massively parallel computer, containing as many as 65,536 processors operating synchronously. Instructions are sent to the processors from a user program running on a front end computer. The CM-2 simulates any number of *virtual processors* by automatically time and memory slicing the activities of the physical processors.

The interface between the user program and the CM-2 is a subroutine library, called Paris, loaded onto the front end. We used Paris version 5.0 in our development. Each Paris call results in a macro-instruction being sent to the CM-2 which is then translated into lower level instructions that the processors understand.

The physical processors of the CM-2 are connected in a hypercube communication network. The CM-2 provides a clean interface to the hypercube by automatically simulating a *virtual processor grid* of some dimension between 1 and the dimension of the hypercube. The user defines the number of dimensions and size of each dimension. In our applications we use only one and two dimensional virtual grids. Processors are laid out in the virtual grid using a gray code mapping [7, 14, 18], which insures that processors with neighboring grid coordinates are connected by an actual communication link.

There are three forms of interprocessor communication available: *grid neighbor* communication, *router* communications, and *scans*. A grid neighbor communication allows virtual processors which are geometrically adjacent in the virtual processor grid to exchange data. Grid neighbor communication occurs quickly, in constant time, because of the communication link guaranteed by the grid layout. The router is a very general communication mechanism which allows various communication patterns on the machine, such as global reductions and single-datum broadcasts. In this paper, we shall be concerned only with the most complex of these communication patterns, router *permutations*, which allow processors to simultaneously send and receive data from any other arbitrary processor in the machine. Router permutations are very slow but allow the programmer to ignore the topology of the virtual grid in data movements.

Scans are intermediate in speed and perform parallel prefix or postfix computations along with the communication. They are defined as follows follows: Suppose we have an ordered class of processors, p_1, p_2, ..., p_k, (called a "scan class") each containing a datum d_i. Then the result of a prefix (resp., postfix) computation at processor p_j is $\sum_{1 \le i \le j} d_i$ (resp., $\sum_{j \le i \le k} d_i$) where $\sum$ is some associative operator. Examples are $+$, $\times$, `max`, `min`, and "copy d_1". A scan class of processors is determined by specifying some dimension of the current grid configuration described above. For example, if the grid is two dimensional, we may scan along rows or columns. The results of a segmented scan through the rows and a non-segmented scan through the columns of a two-dimensional virtual processor grid are illustrated in Figure 2. The CM-2 scan functions

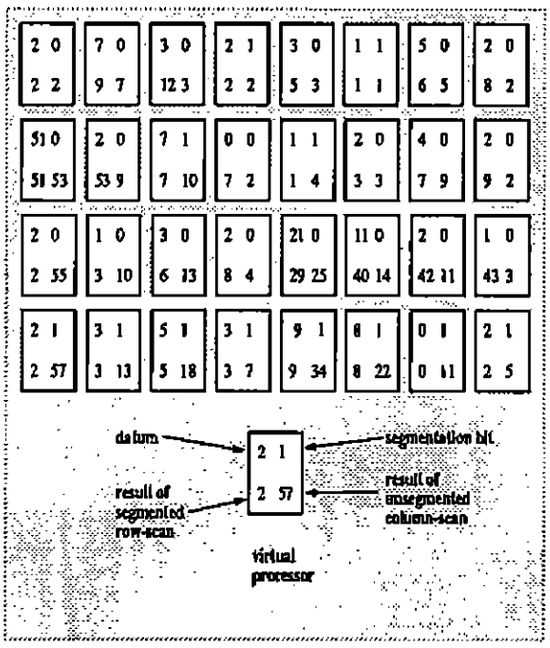


Figure 2: Example of Scans in a 2-D Virtual Processor Grid.

also allow postfix sums and "segmented" scans. The segmented scans allow the summation to be zeroed at any virtual processor in the scan class having a "segmentation bit" set. By combining communication with computation, the CM-2 scan() primitives operate very fast. In fact, previous researchers [4] have assumed the time for a scan to be $O(1)$.

2.1 Complexity of Operations

We use the following hierarchy in analyzing the time complexity of our algorithms:

Speed	Time	Operation
fast	1 step	local computation (add,mult,etc.) grid-neighbor communication
medium	$\mathcal{S}$ steps	Scans
slow	$\mathcal{R}$ steps	Router Permutations

This hierarchy is chosen to be reflective of the Paris programming interface. We view the CM-2 as some virtual machine executing the Paris instruction set, rather than as a raw SIMD hypercube.

If we were given direct access to the hypercube channels and the microcode of the data processors and sequencer, several of our operations involving data movements could be done much faster. In practice, however, such access in not generally available nor is it practical. We caution the reader, then, that there is a great deal of information hiding behind the $\mathcal{R}$ and $\mathcal{S}$ symbols. Actual time complexities may depend upon many factors, such as the ratio of virtual to physical processors (and thus the problem size), the particular mapping between the grid and the hypercube, the number of active processors, and the degree of scan segmentation, to mention but a few. The implementation of the Paris operations might also be changed or augmented. For example, we could foresee the implementation of a medium-cost hybrid grid-router communication method, in which generalized data communications would be allowed, but only in some dimension of the grid. This would change the nature of $\mathcal{R}$, and could have applicability in the algorithm of Section 4.

3 Algorithm 1: A Row-Sweep Approach

The idea behind the row-sweep approach is to configure the CM-2 as a one dimensional array of processors and then use this array to process the grid graph row by row. Previous CM-2 algorithms [5, 11] have used one-dimensional approaches but with a diagonal-wise sweep through the graph. Diagonal-sweep algorithms have the advantage of requiring only local data communication. However, they can utilize no more processors at any one time than the number of vertices in the current diagonal. This means that computation in the upper left and lower right corners is done with many processors idle. (see Figure 3). In square grid graphs, most of the computation would be done with idle processors. This under-utilization of processors is even more pronounced in cases of long and narrow grid graphs – where M is very small (a few hundred or a few thousand characters) but N is large (hundreds of thousands or millions of characters). This is because at most M processors can simultaneously be in use. (The algorithm of [11] is able to ameliorate this under-utilization in the case of sequence database comparisons).

The row-sweep approach avoids the processor utilization problems of diagonal-sweep approaches. The main difficulty with the row-sweep method is that it introduces transitive dependencies which diagonal-wise methods do not. Fortunately, this apparent drawback will be overcome by carefully re-formulating each row computation, in a manner that enables the use of the fast scan primitives built into the CM-2.

The row-sweep algorithm configures the CM-2 as a one dimensional processor array of length $N + 1$ and then sweeps down the grid graph, associating the processor array with one row of the grid graph at a time. We will show how each row can be processed with a single scan and a few constant-time operations – requiring $O(\mathcal{S})$ time (recall that $\mathcal{S}$ is the time required for a scan operation). Since there are M rows, this algorithm operates in $O(M\mathcal{S})$ total time steps. Note that

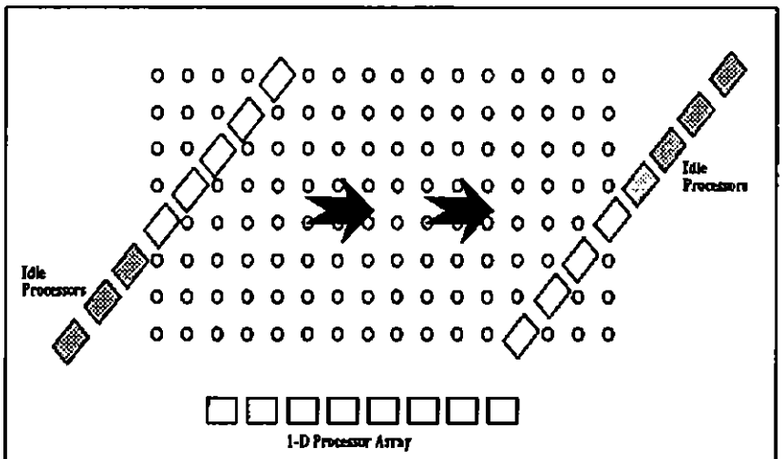


Figure 3: Idle Processors in Diagonal Sweep Algorithms. Here, an 8-processor CM-2 is being used to sweep through an 8×16 grid graph. An average of 30 percent of the processors are idle at each step. This is even worse when the row size of the grid graph falls below the physical processor count, for then at most M processors are ever used.

the row-sweep algorithm is actually a one-dimensional recursive algorithm – even though the scans are treated as primitive their their implementation is recursive.

The algorithm proceeds in M stages, the ith stage consisting of processing row i – computing the cost of a minimum cost path from vertex $(0,0)$ to each vertex (i,j), $0 \leq j \leq N-1$. Let us consider the row number i to be fixed and concentrate upon the computations necessary for this row, assuming we have already processed row $i-1$ at the previous stage. We wish to compute the numbers $C(i,j)$, $j = 0, ..., N-1$. Define

$$r_j = \min\{C(i-1,j-1) + S(y_i, x_j), C(i-1,j) + I(y_i)\}.$$

Thus, the number r_j is the cost of a shortest path from vertex $(0,0)$ to vertex (i,j) that enters vertex (i,j) either diagonally or vertically. Also, define $s_j = C(i,j-1) + D(x_j)$. This is the cost of a shortest path from $(0,0)$ to (i,j) that enters (i,j) horizontally. Obviously, $C(i,j) = \min\{r_j, s_j\}$. Unfortunately, s_j depends upon $C(i,j-1)$, which we are trying to compute simultaneously! However, the definition of s_j implies the following:

$$C(i,j) = \min_{0 \leq k \leq j} \{r_k + \sum_{l=k+1,j} D(x_l)\},$$

because $r_k + \sum_{l=k+1,j} D(x_l)$ is the cost of the path that first reaches row i at vertex (i,k) and proceeds to vertex (i,j) horizontally. This expression is reminiscent of a prefix sum. However,

the CM-2 does not have a scan() function to compute such an expression simultaneously for all $j \in \{0, 1, \ldots N-1\}$. Thus we re-write this formula in a fashion which will prove more suitable. Adding $\sum_{l=j+1,N} D(x_l)$ to both sides of the previous equation gives

$$C(i,j) + \sum_{l=j+1,N} D(x_l) = \min_{0 \le k \le j} \{r_k + \sum_{l=k+1,N} D(x_l)\}.$$

Let us denote by p_k the sum $\sum_{l=k+1,N} D(x_l)$. Thus, p_k is a post-fix sum and can be computed by a scan() function, in parallel for all k. In fact, this expression does not depend upon the row number i, and can be computed and stored once and for all at the beginning of the computation (this last comment is not true of grid graphs that are not derived from sequence comparison problems, in which case we do have to use a scan() for the p_k's of each particular row).

To actually compute $C(i,j)$, it appears we should subtract p_j from the expression above. To avoid this subtraction we change our goal from computing $C(i,j)$ $\forall j$, $0 \le j < N$ to that of computing $C(i,j) + p_j$. In this spirit, let us denote

$$C'(i,j) = C(i,j) + p_j,$$
$$r'_j = r_j + p_j.$$

Observe that $C(i, N-1) = C'(i, N-1)$ (because $p_{N-1} = 0$) and in particular $C(M-1, N-1) = C'(M-1, N-1)$. Note that

$$r'_j = \min\{C'(i-1, j-1) + (S(y_i, x_j) - D(x_j)), C'(i-1, j) + I(y_i)\}, \quad (*)$$
$$C'(i,j) = \min_{0 \le k \le j} r'_k. \quad (**)$$

These two formulas will give the algorithm for computing each row.

3.1 The Algorithm

Initialization:

Assume the CM-2 is configured as a one-dimensional virtual processor grid of length $N+1$ (See Section 2) and each processor is loaded with a copy of the (small) weight tables I, S, and D. The initial C' values are all the same and are easy to compute: $C'(0,j) = \sum_{l=1,N} D(x_l)$. This initialization requires only a single scan and constant time operations.

For each row i:

Processor j will compute $C'(i,j)$. First it computes r'_j as follows: The number $C'(i-1,j)$ is locally available from the previous step, and $C'(i-1, j-1)$ is available from the neighboring processor $(i, j-1)$ by using a grid communication (Section 2). The front end broadcasts the character y_i (in $O(1)$ time) to all the processors, so that the weights

$I(y_i)$ and $S(y_i, x_j) - D(x_j)$ can be obtained from a local table lookup. Then r'_j is computed according to formula (*) in constant time. Once the r'_js are computed we compute $C'(i,j)$ with a prefix-minimum scan according to formula (**). Thus the computation for a single row requires only a single scan and constant time operations.

Summing up, the overall time complexity is $O(M\mathcal{S})$.

3.2 Performance Results

In this section we give timing results from our implementations. The CM-2 measurements were made on the unit in operation at the Numerical Aerodynamic Simulation Facility of NASA Ames Research Center, Mountain View, CA. This unit is configured with 32,768 processors although we only used 8,192, the remaining processors being dedicated to other projects. These facilities were graciously made available by the NASA Research Institute for Advanced Computer Science (RIACS).

The tables in Figure 3.2 summarize timing comparisons made between the row-sweep algorithm running on the CM-2, the sequential algorithm running on a Sun 4/110 computer and a diagonal sweeping algorithm on an Alliant FX/8 computer. These are the "best" machines we had available, we feel, for the various sequence comparison algorithms. These comparisons are simplistic, and are given in the interest of contrasting algorithms, not machines. The Alliant FX/8 is a vector mini-supercomputer. The row-sweep and diagonal-sweep methods are both "vector-like" approaches to the problem. The row-sweep method is amenable to the CM-2, because vector lengths are large and static and the CM-2 is adept at handling the transitive dependencies in each row. The diagonal-sweep method does not introduce transitive dependencies but is most amenable to machines which may dynamically expand and shrink vector lengths, such as the Alliant FX/8. The Sun 4/110 is a sequential computer, roughly rated a speed of 7 to 8 million instructions per second (MIPS), and is most commonly used as a scientific workstation.

The timing results for both the Alliant FX-8 and the Sun 4/110 scale linearly with the problem size, and we use them as a baseline for observing non-linearities in the behaviour of the CM-2. The CM-2 speedups increase as row size becomes longer, allowing the communication overhead of scan primitives to be more widely amortized over more virtual processors. This non-linearity does not exist along column sizes. The CM2 is as efficient at processing a grid graph with a small number of rows as it is for one with many. It is important that efficiency not be retarded by small M, because of the important cases arising in applications that yield a long and thin grid graph, such as few-to-many database comparisons (see Section 1.5).

string lengths ($\|X\|,\|Y\|$)	SUN 4/110 sequential alg		ALLIANT FX/8 diagonal sweep alg		8192 pe CM-2 row sweep alg	
	time	(CM-2 speedup)	time	(CM-2 speedup)	time	V/P ratio
(16384,100)	28	(127)	7.8	(35.5)	.22	2
(32768,100)	50	(172)	16.5	(56.9)	.29	4
(65536,100)	90	(200)	32.9	(73.1)	.45	8
(131072,100)	179	(229)	62.3	(79.9)	.78	16
(262144,100)	345	(246)	129	(92.1)	1.4	32

string lengths ($\|X\|,\|Y\|$)	SUN 4/110 sequential alg		ALLIANT FX/8 diagonal sweep alg		8192 pe CM-2 row sweep alg	
	time	(CM-2 speedup)	time	(CM-2 speedup)	time	V/P ratio
(16384,8192)	2262	(129)	631	(36.1)	17.5	2
(32768,8192)	4041	(174)	1287	(55.5)	23.2	4
(65536,8192)	7385	(201)	2607	(70.8)	36.8	8
(131072,8192)	14665	(230)	5307	(83.3)	63.7	16
(262144,8192)	28299	(240)	10849	(91.9)	118	32

Figure 3.2: Timings of the Row-Sweep Method. The speedup column is the ratio of the comparison machine time over the CM-2 time. V/P is the ratio of virtual to physical processors. All times are in seconds.

4 Algorithm Two: A Block-Recursive Approach

The focus of the block-recursive algorithm is to maximize the processor usage. The CM-2 is configured as a two-dimensional M by N virtual processor grid, the (i,j) virtual processor being associated with the (i,j) vertex of the grid graph. Thus the algorithm uses $O(NM)$ processors – far more than the preceeding ones. It operates in

$$O((\mathcal{S}+\mathcal{R})\log N \log M)$$

time steps. The block recursive algorithm solves a more general problem than the single source-sink path length:

Compute the lengths of shortest paths between **all** *boundary vertices of the grid graph.*

We shall refer to a collection of data values distributed across a 2-D virtual processor grid as a *matrix* of values. The $[i,j]$ entry of a matrix M is denoted $M[i,j]$ and is stored in the (i,j) processor of the grid. The left, top, bottom, and right boundaries of grid graph G are denoted L_G, T_G, B_G, and R_G, respectively, and are ordered as in Figure 4. Also, the notation $\alpha_H\beta_K$ will denote a *distance matrix* containing the lengths of shortest paths from the α boundary of grid graph H to the β boundary of grid graph K ($\alpha,\beta \in \{\mathtt{L},\mathtt{T},\mathtt{R},\mathtt{B}\}$). Given a grid graph G, then, the goal is to compute the four distance matrices T_GR_G, T_GB_G, L_GR_G, and L_GB_G. Since the grid graph G is $k \times k$, these matrices are also $k \times k$ and are stored in the processors of G.

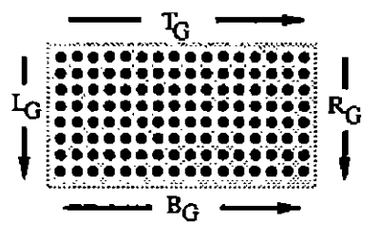


Figure 4: Ordering of boundary vertices.

4.1 Initialization

We assume the algorithm starts with source (resp. destination) string characters loaded onto the processors assigned to the top (resp. left) boundary of the virtual processor grid. Throughout the algorithm, the (i,j) processor will hold the values $D(x_j)$, $I(y_i)$, and $S(y_i,x_j)$ in its local memory. These values may be established in time bounded by $O(\mathcal{S})$ (The details are easy and are omitted).

4.2 The Recursion.

The block-recursive algorithm processes the whole graph at once, instead of concentrating on one row or diagonal at a time. In what follows we will assume the grid is square – the non-square case is very similar. The recursion is not new. It is used in several theoretical results – [2, 9, 17] for example. We will show how to carry it out using the CM-2 primitives.

- *Input:* An $M \times M$ grid graph G with one processor per vertex. The processor assigned to vertex (i, j) contains the edge weights $I(y_i)$, $D(x_j)$, and $S(y_i, x_j)$.
- *Output:* The four distance matrices $T_G R_G$, $T_G B_G$, $L_G R_G$, and $L_G B_G$ stored in G.

- *Step 0:* If G is a single node then return the 1×1 distance matrix [0].
- *Step 1:* Divide the grid graph into four equal quadrants, A, B, C, and D as in Figure 4.2. Each quadrant is itself a valid grid graph with one processor per vertex. In parallel, recursively compute boundary-boundary path information for the four subgraphs. After these (parallel) recursive calls return, the distance matrices $T_A R_A$, $T_A B_A$, $L_A R_A$, and $L_A B_A$ are available in the processors of quadrant A. The same holds for quadrants B, C, and D.
- *Step 2.1:* Using processors in quadrants A and B, **merge** the distance matrices of A and B to obtain the distance matrices for $A \cup B$: $T_{A \cup B} R_{A \cup B}$, $T_{A \cup B} B_{A \cup B}$, $L_{A \cup B} R_{A \cup B}$ and $L_{A \cup B} B_{A \cup B}$.
- *Step 2.2:* Using processors in quadrants C and D, **merge** the distance matrices of C and D to obtain the distance matrices for $C \cup D$: $T_{C \cup D} R_{C \cup D}$, $T_{C \cup D} B_{C \cup D}$, $L_{C \cup D} R_{C \cup D}$ and $L_{C \cup D} B_{C \cup D}$.
- *Step 2.3:* Using all the processors of G, **merge** the distance matrices of $A \cup B$ and $C \cup D$ to obtain the distance matrices for G: $T_G R_G$, $T_G B_G$, $L_G R_G$, and $L_G B_G$.

Steps 2.1 and 2.2 are carried out in parallel. The distance matrices for $A \cup B$ and $C \cup D$ are no larger than $k \times k$ and are stored in temporary locations in G.

The time complexity for steps 2.1 – 2.3 is dominated by the merge procedures – there are some router calls necessary for inter-quadrant data movements but these are only minor housekeeping matters. The time complexity $T(k)$ of the recursion, then, for a $k \times k$ grid graph, obeys the equation

$$T(k) \leq T(\frac{k}{2}) + c \cdot \text{Merge}(k)$$

where Merge(k) represents the time to perform the merging step on two $k \times k$ grid graphs and c is some constant. Thus, the overall time requirements for an $M \times M$ grid graph is bounded by

$$O(\log M \text{Merge}(M)).$$

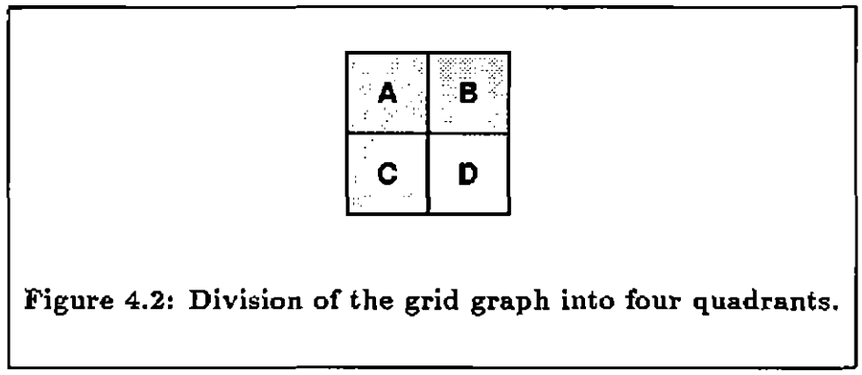


Figure 4.2: Division of the grid graph into four quadrants.

In the next section we will show that Merge(k) is bounded by $O((\mathcal{S}+\mathcal{R})\log k)$ time steps.

4.3 The Merge Step

We will describe the merge of quadrants A and B – the other cases are nearly identical. The matrices $T_{A\cup B}R_{A\cup B}$, $T_{A\cup B}B_{A\cup B}$, $L_{A\cup B}R_{A\cup B}$, and $L_{A\cup B}B_{A\cup B}$ will be the composition of smaller ones:

$$\left[\ T_{A\cup B}B_{A\cup B}\ \right] = \begin{bmatrix} T_AB_A & T_AB_B \\ T_BB_A & T_BB_B \end{bmatrix}, \quad \left[\ T_{A\cup B}R_{A\cup B}\ \right] = \begin{bmatrix} T_AR_B \\ T_BR_B \end{bmatrix}$$

$$\left[\ L_{A\cup B}B_{A\cup B}\ \right] = \begin{bmatrix} L_AB_A & L_AB_B \end{bmatrix}, \quad \left[\ L_{A\cup B}R_{A\cup B}\ \right] = \begin{bmatrix} L_AR_B \end{bmatrix}$$

There are four submatrices that must be computed: T_AR_B, T_AB_B, L_AR_B, and L_AB_B. The others are either already computed, such as T_AB_A or are the infinity matrix [∞], such as T_BB_A. We shall show how to compute one of these sub-matrices, T_AR_B. The remaining matrices are handled identically.

The (i,j) entry of T_AR_B should be the length of a shortest path from the ith vertex on T_A to the jth vertex on R_B – a path which must pass through a vertex on R_A, then through an A-to-B connecting edge to a vertex on L_B. The first step is to "augment" the matrix L_BR_B with information from the A-to-B connecting edges so that it becomes R_AR_B. The connecting edge information is found on processors assigned to L_B vertices. This information is distributed to all of B using a permutation route (all inside G). A few constant time operations suffice to compute R_AR_B – altogther at most $O(\mathcal{R})$ time steps (The details are omitted).

By copying the augmented matrix R_AR_B from quadrant B to A we now have the matrices T_AR_A and R_AR_B stored in quadrant A. The entries of the solution matrix, T_AR_B, are given by the formula

$$T_AR_B[i,j] = \min_{0\le h\le k/2-1}\{T_AR_A[i,h] + R_AR_B[h,j]\}. \qquad (1)$$

There may be more than one value of h which minimizes the right hand side of the above equation; let $\theta(i,j)$ denote the largest such h. We point out a very important property about the θ values: they are ordered along rows; i.e.

$$p \leq q \Longrightarrow \theta(i,p) \leq \theta(i,q). \tag{2}$$

The relation in formula (2) is well known – we call it the *path ordering property*. It is a reflection of the fact that the θ values are indices of vertices where shortest paths cross the R_A boundary. The proof of (2) is trivial and is omitted. Formula 1 says that the matrix $T_A R_B$ is obtained by "multiplying" the matrices $T_A R_A$ and $R_A R_B$ in the closed semi-ring $(\min, +)$. This special multiply will be carried out by the processors in quadrant A. We simultaneously use other quadrants for other parallel multiplies – for example we use quadrant B to compute $T_A B_B$.

The time for a merge is dominated by the multiplication algorithm, given below, which runs in $O((\mathcal{S}+\mathcal{R})\log k)$ time. Since the augmentation and matrix copy operations require $O(\mathcal{R})$ time steps, the overall time complexity for a merge is $O((\mathcal{S}+\mathcal{R})\log k)$.

4.3.1 Matrix Multiplication

The matrix multiplication subroutine takes $k \times k$ matrices U and V and outputs a product matrix $T = U * V$ defined by the formula

$$T[i,j] = \min_{0 \leq h \leq k} \{U[i,h] + V[h,j]\}. \tag{3}$$

As above,

$$\theta(i,j) = \max\{h | T[i,j] = U[i,h] + V[h,j]\} \tag{4}$$

and thus

$$T[i,j] = U[i,\theta(i,j)] + V[\theta(i,j),j]. \tag{5}$$

The multiplication algorithm assumes θ satisfies property (2). It proceeds in three stages:

Stage M1: Compute θ implicitly. This requires $O((\mathcal{S}+\mathcal{R})\log k)$ time steps.
Stage M2: Compute θ explicitly. This requires $O(\mathcal{S}+\mathcal{R})$ time steps.
Stage M3: Use θ to compute T. This requires $O(\mathcal{R})$ time steps.

Thus, the time complexity of the multiply is $O((\mathcal{S}+\mathcal{R})\log k)$. In what follows we describe the action for the ith row – it is understood that the same actions take place simultaneously in all rows (and in all subgrids). Also, given some index set $S \subseteq \{0,1,2,\ldots k-1\}$, by "computing $\theta(i,S)$", we mean computing $\theta(i,s)$ for all $s \in S$.

Stage M1: Computing θ implicitly.

This is the most complicated of steps M1 – M3. It is performed by computing, in implicit form, $\theta(i,S)$ for a succession of larger and larger sets S: in particular $\theta(i,S_0)$, $\theta(i,S_1)$, ..., $\theta(i,S_{\log k})$, where $S_0 = \{k/2\}$, $S_1 = \{k/4, k/2, 3k/4\}$, $S_2 = \{k/8, k/4, 3k/8, k/2, 5k/8, 3k/4, 7k/8\}$, ..., $S_{\log k} = \{0, 1, 2, 3, \ldots, k-1\}$. The goal of stage M1 is to compute $\theta(i, S_{\log k})$.

$\theta(i,S)$ is stored implicitly over the ith row as follows: The path ordering property (2) implies that S is actually divided into non-overlapping ranges of indices for which θ values are the same; i.e., for each $j \in \theta(i,S)$ there are range limits $p, q \in S$ such that $\{s \in S | \theta(i,s) = j\} = [p,q] \cap S$. In Figure 4.3.1(a), for example, S_2 could be divided into the index ranges $[2,2]$, $[4,8]$, $[10,12]$, and $[14,14]$. We represent $\theta(i,S)$ implicitly by having processor (i,j) store the range limits p and q in temporary matrices P and Q respectively. Of course, the range may be empty – this is indicated by an additional temporary bit matrix CROSSED, where CROSSED$[i,j]$ is cleared or set according to whether $\{s \in S | \theta(i,s) = j\}$ is empty or not.

We now describe how $\theta(i, S_{\log k})$ is computed. This requires $\log k + 1$ steps, numbered 0 to $\log k$, At the end of step h, S_h is computed. Each step runs in $O(\mathcal{S} + \mathcal{R})$ time. Thus, the entirety of stage M1 of the multiplication requires $O((\mathcal{S} + \mathcal{R}) \log k)$ time, as claimed above. Note that each index $\gamma \in S_h - S_{h-1}$ is sandwiched between two indices α and β in S_{h-1}, and in fact $\gamma = (\alpha + \beta)/2$. Property (2) says that $\theta(i,\gamma)$ is also sandwiched: $\theta(i,\alpha) \leq \theta(i,\gamma) \leq \theta(i,\beta)$, so

$$T[i,\gamma] = \min\{U[i,l] + V[l,\gamma] \mid \theta(i,\alpha) \leq l \leq \theta(i,\beta)\}.$$

Thus, it suffices to have all processors (i,l), where $\theta(i,\alpha) \leq l \leq \theta(i,\beta)$, to hold a contest to see which value of l will be $\theta(i,\gamma)$. The contest goes as follows: Each processor (i,l) can determine the index γ by scanning the Q values rightward and the P values leftward, taking the average, then storing it in a temporary matrix GAMMA. These scans are segmented at the CROSSED processors. A router permutation communication (all inside G) is used to obtain $V[l,\gamma]$ – this is added to $U[i,l]$ (available locally) to get the TRIAL_VALUE $U[i,l] + V[l,\gamma]$. A minimum scan determines the winner. Segmentation processors (previous winners) must participate in two scans. Winners update their P and Q entries accordingly and set the segmentation bit in the CROSSED matrix. This process is illustrated in Figure 4.3.1(b).

Stage M2: Computing θ explicitly.

In this stage the actual θ values will be computed and stored into a temporary matrix THETA. This matrix is filled as follows: First, all THETA entries are initialized to zero. If the flag bit CROSSED$[i,j]$ is set, processor (i,j) writes j into the THETA matrix entry of processor $(i, \mathtt{P}[i,j])$; i.e. processor (i,j) sets THETA$[i, \mathtt{P}[i,j]] = j$. This requires a router step. The THETA indices are then replicated rightward with a max-scan – this sets THETA$[i, \mathtt{P}[i,j]+1] =$ THETA$[i, \mathtt{P}[i,j]+2] = \ldots =$ THETA$[i, \mathtt{Q}[i,j]] = j$ as desired. In summary, THETA can be obtained from the implicit range form in $O(\mathcal{S} + \mathcal{R})$ time steps.

$$S_2 = \{2,4,6,8,10,12,14\}$$

$$\theta(i, S_2) = \{1,4,4,4,12,12,13\}$$

CROSSED	[	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0	]
P	[	-	2	-	-	4	-	-	-	-	-	-	-	10	14	-	16	]
Q	[	0	2	-	-	8	-	-	-	-	-	-	-	12	14	-	-	]

Figure 4.3.1(a): Implicit Storage Scheme. Here, k = 16. The ith row of the matrices P, Q, and CROSSED constitute an implicit storage of $\theta(i, S_2)$.

⟶ ⟶ scan on Q ⟶ ⟶

scan result	[	0	2	2	2	2	8	8	8	8	8	8	8	12	14	14	14	]

⟵ ⟵ scan on P ⟵ ⟵

scan result	[	2	4	4	4	10	10	10	10	10	10	10	10	14	16	16	16	]

GAMMA	[	1	1,3	3	3	3,9	9	9	9	9	9	9	9	9,13	13,15	15	15	]
TRIAL_VALUE	[	8	2,5	9	6	7,9	4	3	7	3	5	9	4	7,4	3,4	4	4	]
CROSSED	[	0	1	0	0	1	0	0	0	1	0	0	0	1	1	0	1	]
P	[	-	1	-	-	4	-	-	-	9	-	-	-	10	13	-	15	]
Q	[	-	3	-	-	8	-	-	-	9	-	-	-	12	14	-	15	]

Figure 4.3.1(b): Computing $\theta(i, S_3)$. The TRIAL_VALUEs are hypothetical. Note that segmentation processors (where CROSSED = 1) must participate in two contests, so they compute two γ values and two trial values.

CROSSED	[	0	1	0	0	1	0	0	0	1	0	0	0	1	1	0	1	]
P	[	-	0	-	-	4	-	-	-	9	-	-	-	10	13	-	15	]
Q	[	-	3	-	-	8	-	-	-	9	-	-	-	12	14	-	15	]

after index inscription:

THETA	[	1	0	0	0	4	0	0	0	0	8	12	0	0	13	0	15	]

after max-scan:

THETA	[	1	1	1	1	4	4	4	4	4	8	12	12	12	13	13	15	]

Figure 4.3.1: Stage M2 – Computing θ explicitly. Step 4 of stage M1 has been completed; i.e. $\theta(i, S_4)$ is computed. The index inscription requires a router permutation within the matrix. The last step requires a scan.

Stage M3: Computing the result T.
Once THETA[i,j] is known, the (i,j) processor can immediately compute $T[i,j]$ by obtaining the U value from processor $(i, \texttt{THETA}[i,j])$ and the V value from processor $(\texttt{THETA}[i,j], j)$ and then adding them as in formula (5). This is done with two router permutation communications (all inside G) and a few constant time operations – $O(\mathcal{R})$ time.

Only a few test cases could be run because the method consumes a quadratic number of virtual processors. An 8192 processor CM-2 solved the problem for a 128×128 grid graph in 7.4 seconds, and for a 256×256 grid graph in 30.7 seconds. This algorithm is drastically lower in its asymptotic complexity than the row-sweep method, but is slower for small problem sizes. The main problem, we feel, is the necessity of using utterly global routes and scans in situations where only subgrid-specific routes and scans are needed. However, this program is a viable candidate for use in CM-2 configurations that are large (relative to the length of the sequences being compared) and processor utilization becomes a problem.

5 Summary and Future Work

We have measured our sequence comparison algorithms in terms of their "prefix complexity" – a count of the number of calls to functions computing a parallel prefix or postfix, such as those in the

scan() family. The dominant theme to our solution to the sequence comparison problem on the CM-2 has been the use these highly optimized scan() operators. We have used these primitives to implement two algorithms for sequence comparisons. The first operates well when the problem size is large with respect to the processor count of the CM-2 being used. The second algorithm is more complicated but is polylogarithmic in its asymptotic running time. Although the timing results make it appear worse than the first algorithm, it may become important in future versions of the CM-2, which may have a vastly greater number of processors than today's versions. In such a situation, the processor count would dominate problem sizes, and a routine that runs in polylogarithmic time would be superior.

There are several open questions, which are not necessarily germane to sequence comparison, that we have touched upon in this paper. The first is the question of determing the theoretical "prefix complexity" for a wider range of problems. Blelloch [4] has investigated the complexity of designing several algorithms with these primitives under the assumption that scan operations require only a constant amount time (we have chosen to distinguish them from constant time operations).

Secondly, we have expressed some architectural desires. The notion of "grid" pervades the CM-2 architecture. Grid facilities could be enhanced and extended even further. It would be nice to establish a hybrid grid-router communication mechanism which would have performance somewhere between that of the grid-neighbor mechanism (NEWS mechanism in CM-2 lingo), which is very fast, and the generalized router mechanism, which is very slow. Such a mechanism might allow medium-speed communication between arbitrary processors, but only in some particular dimension of the currently defined grid, or in some other well defined non-dimensional partition of the grid. Such a partition would be the square sub-grids of the block recursive algorithm. This might also speed up scans by basing them on such partitions, rather than on arbitrary segmentation bits (although scans are already quite fast). Then, the intra-grid communication could take place in a limited, and hopefully faster, fashion. This may improve the performance of the block recursive algorithm, which relies upon scans and router communication in each recursive sub-grid. We have also noted the disadvantage of the vector length sensitivity of the CM-2. Grid stretching and shrinking operations, with user defined transformation functions, might be handy.

References

[1] A. AGGARWAL AND J. PARK. Notes on Searching in Multidimensional Monotone Arrays. *Proc. 29th Annual Symp. on Foundations of Computer Science*, pp. 497-512.

[2] A. A. APOSTOLICO, M. J. ATALLAH, L. L. LARMORE, S. MCFADDIN. Efficient Parallel Algorithms for String Editing and Related Problems, *SIAM Journal on Computing*, October 1990.

[3] BILOFSKY, H.S. AND BURKS, C. The GenBank (R) Genetic Sequence Data Bank, *Nucleic Acids Research*, 16: 1861-1864, (1988).

[4] G. E. BLELLOCH. Scan Primitives and Parallel Vector Models, *Ph.D. Thesis, Massachusetts Institute of Technology*, (1988).

[5] E. EDMISTON AND R. A. WAGNER. Parallelization of the Dynamic Programming Algorithm for Comparison of Sequences, *Proceedings of 1987 International Conference on Parallel Processing* pp. 78-80 (1987).

[6] H. FUCHS, Z.M. KEDEM, AND S.P. USELTON. Optimal Surface Reconstruction from Planar Contours, *Communications of the ACM*, 20, pp. 693-702, (1977).

[7] E. N. GILBERT. Gray Codes and Paths on the N-cube, Bell System Tech. J., 37, p. 915 (1958).

[8] W.D. HILLIS. The Connection Machine, *The MIT Press*, (1985).

[9] O.H. IBARRA, T. PONG, S.M. SOHN. Hypercube Algorithms for Some String Comparison Problems, *Unpublished Manuscript*, (January 1988).

[10] R.E. LADNER AND M.J. FISCHER. Parallel Prefix Computation, *Journal of the ACM* 27, 4, pp. 831–838 (1980).

[11] E. LANDER, J. P. MESIROV, W. TAYLOR IV. Protein Sequence Comparison on a Data Parallel Computer, *Proceedings of 1988 International Conference on Parallel Processing* pp. 257-263 (1988).

[12] H. M. MARTINEZ (ED.) Mathematical and Computational Problems in the Analysis of Molecular Sequences, *Bulletin of Mathematical Biology* (Special Issue Honoring M. O. Dayhoff), 46, 4 (1984).

[13] T. R. MATHIES. A Fast Parallel Algorithm to Determine Edit Distance, *Unpublished Manuscript* (April 1988).

[14] O.A. MCBRYAN, E. F. VAN DE VELDE. Hypercube Algorithms and Implementations, *SIAM Journal on Scientific and Statistical Computing* Vol. 8, No. 2 (March 1987).

[15] 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, pp.443–453 (1973).

[16] A. GIBBONS AND W. RYTTER. Efficient Parallel Algorithms, *Cambridge University Press*, New York (1988).

[17] S. RANKA AND S. SAHNI. String Editing on an SIMD Hypercube Multicomputer, *Journal of Parallel and Distributed Computing*, vol. 9, No. 4, pp. 411-418 (1990).

[18] J. SALMON. Binary Gray Codes and the Mapping of a Physical Lattice into a Hypercube, Caltech Concurrent Processor Report (CCP) Hm-51 (1983).

[19] D. SANKOFF AND J. B. KRUSKAL (EDS.). Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison, *Addison-Wesley*, Reading, PA (1983).

[20] P.H. SELLERS. The Theory and Computation of Evolutionary Distance: Pattern Recognition, *Journal of Algorithms* 1, pp.359–373 (1980).

[21] R. A. WAGNER AND M. J. FISCHER. The String to String Correction Problem, *Journal of the ACM* 21,1, pp.168–173 (1974).