Approximate String Matching on the DADO2 Parallel Computer[1]

Toshikatsu Mori [2]
and
Salvatore J. Stolfo

Department of Computer Science
Columbia University
New York, NY 10027

CUCS-361-88
January 22, 1988

Abstract

This paper presents an approximate string matching algorithm on the 1023-processor parallel computer DADO2. To allow proximity in matching between the text and search pattern, the dynamic programming method is used as the matching algorithm. This paper includes timing measurements and comparison with a conventional sequential computer (VAX). The results show significant speedup over the sequential computer.

1. This research was conducted as part of DADO project. It has been supported by the Defense Advanced Research Projects Agency under contract NOOO39-84-C-0165, and the New York State Science and Technology Foundation under contract NYSSTFCAT(84)-15, as well as by grants from DEC, Hewlett Packard, IBM, AT&T, Intel, and Valid Logic Systems.

2. Visiting from Processor Engineering Department, Second Office Automation Division, NEC Corporation. *Address*: 1-10, Nisshincho, Fuchu, Tokyo, 183 Japan. ***Phone number***: 011-81-423-33-1545

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

The purpose of *approximate string matching* (see for example [1]) is to find instances of a search pattern in a symbol structure where some *differences* between the pattern and the structure are allowed. It is useful in "free text" retrieval systems that must be sensitive to typographical errors and abbreviations. It is also of great practical importance in speech recognition systems that match the symbolic output of an acoustic processing step against many potential utterances. The acoustic processes are typically error prone, admitting representations that approximate the actual utterance. Hence the search process must allow for omitted, extraneous, or transformed symbol structures (see [2] for examples).

The common characteristic of these problems is clear; the search pattern cannot be found simply by direct character comparisons and searching must allow for a number of *differences* when comparing the pattern to various strings within the text. This characteristic invalidates the efficient approaches to searching that depend upon static data structures (inverted files of indices of symbols appearing in the text, for example). The best known approaches to solving this problem are based upon completely scanning the text using *dynamic programming* as the primary search strategy. In this paper we consider the parallelization of this task and measure the effective speedup that parallelism offers in the solution to this problem.

2. Approximate String Matching

Approximate string matching may be demonstrated by the following example.

Consider the problem of finding matches in the text "COLUMBIA" for the pattern "COLKUBYA". One possible correspondence between the two strings is as follows.

position:	1	2	3	4	5	6	7	8	9
text:	C	O	L		U	M	B	I	A
pattern:	C	O	L	K	U		B	Y	A

Here, we can observe three types of differences:

(a) *Insertion*:	a character of the pattern corresponds to no character in the text. (position 4)
(b) *Deletion*:	a character of the text corresponds to no character in the pattern. (position 6)
(c) *Substitution*:	a character of the pattern corresponds to a different character of the text. (position 8)

We assign a cost of *one* to each type of difference; non-unit cost may also be used as the application demands. Under the unit cost assumption, the above pattern matches the text with three differences. This is the minimal cost match. We use the term *occurrence* to refer to a match, that is, an instance of the pattern in the text with some number of differences. In this paper, we will consider the task of finding all occurrences of a pattern in a text where at most k differences are allowed, for a given integer k. Our primary focus is on the parallelization of this task on the DADO2 parallel computer, described briefly in the next section. See Appendix A for a detailed example actually performed on the DADO2.

3. The DADO Machine and Programming Environment

DADO is a tree-structured parallel machine in which the processing elements (PEs) are interconnected in a complete binary tree [3] [4]. A 1023-PE model named DADO2 has been working at Columbia University since 1985. Although DADO was originally designed as an experimental machine to support AI expert systems implemented in rule-based form, it has been found suitable for the high-speed execution of *almost decomposable searching problems* [5]. This section describes the DADO2 machine, its communication primitives, and the DADO programming environment.

3.1 PE Configuration and Operation Mode. Each PE consists of an Intel 8751 8-bit microprocessor, 16 Kb (kilobytes) random access memory (RAM) and a semi-custom I/O chip for high-speed communication. Each PE contains its control program and user program in its 20 Kb local memory (4 Kb on-chip EPROM and 16 Kb RAM). Most instructions are executed in one microsecond.

Each PE operates in one of two modes: SIMD (Single Instruction stream, Multiple Data stream) or MIMD (Multiple Instruction stream, Multiple Data stream). A PE in SIMD mode receives instructions (function addresses) broadcast by its nearest MIMD root ancestor. A PE in MIMD mode is disconnected from its parent, and executes instructions independently of its ancestors in the DADO tree. If the MIMD PE finds a SIMD block in its program, the PE broadcasts the address of the SIMD block to its SIMD descendants. The enabled SIMD descendants execute this block in parallel with the MIMD PE.

3.2 Communication. The communication among PEs is done through the I/O chips. Data transfer between the I/O chip and local memory is under microprocessor control. The basic communication functions are:

(a) *Broadcast*:	Send information to the descendant PEs,
(b) *Resolve*:	Select one PE from a candidate set,
(c) *Report*:	Send information from the selected PE to the root PE.

The resolve operation is a unique function of the DADO machine. As noted above, the purpose of the resolve is to select a PE from a set of designated PEs. In general, this is accomplished by finding the minimum value among all such values held by the set of designated PEs. The minimum value will reach the MIMD PE, and the PE responsible for that value (and only that PE) will know that it is the "winner" of the resolve.

As with the broadcast and report operations, software functions are built atop this hardware circuit to provide a class of functions for resolving on higher-level values.

3.3 Host interface. The DADO machine functions as an attached processor controlled by a conventional computer. The DADO has been used successfully with a range of hosts. In the work reported here, a DEC VAX 11/750 served as the host processor and was connected to the DADO with a DEC DR11-W parallel interface.

It is interesting to note that the DADO2 is comparable to a VAX 11/750 in size, number of components, and age of their respective technologies. Hence, the VAX/DADO2 configuration should be considered twice the cost of a single VAX.

3.4 Languages. To date, three high-level languages are available on DADO2; parallel PL/M (PPL/M), parallel C, and parallel PSL Lisp (PPSL) [6] [7]. Each parallel language accommodates a small set of parallel processing and communication primitives. The program implemented in this paper was written in parallel C.

4. Implementation of approximate string matching on DADO

The abstract algorithm implemented on DADO is presented in this section. In this implementation, the root PE of the DADO tree is in MIMD mode and the other PEs are in SIMD mode. Hereafter we will use n for text size, m for pattern size, k for the number of differences, and p for the number of PEs used in the DADO.

There are three phases to this implementation of the approximate string matching algorithm. First is the *allocation phase*. The host processor sends both a search pattern and a text string to the DADO. The DADO PEs partition the text among them. Next is the *match phase*. All PEs execute the match algorithm and save pointers to the matches in their local memory. Last is the *report phase*. The root PE collects these pointers and sends them to the host. The following sections describe each of these phases in more detail.

4.1 Allocation phase. The host processor sends the search pattern and text to the root PE of DADO, which then broadcasts both to all PEs. All PEs receive the entire search pattern, but store a different portion of the text.

The text is partitioned into contiguous and overlapped pieces and distributed over all available PEs. The partitioned text in each PE is overlapped with that of the adjacent PEs, since if the text were divided without overlap, a PE could not find the pattern that "straddles" across to the adjacent PEs. By eliminating data dependencies in this manner, we improve performance by avoiding additional communication among the PEs in the match phase. The additional work on the host side and timing penalties are small; they are described below and in Section 4.3.

```
        |------------------------- < text > -------------------------|
pe#1    |------------|
pe#2             |------------|
pe#3                      |------------|
 :
 :
 :
pe#p-1                                      |------------|
pe#p                                                 |------------|
```

Figure 1. Data Partition for PEs

The minimum size of the overlap is $m + k - 1$ per PE, since the result of matching a string starting from position i will always be determined before position $i + m + k$.

The host supports this text partitioning as follows.

First, an occurrence that is wholly contained within the overlapped area between two PEs is reported by both PEs. We leave its handling to the host, though this duplication could be eliminated in DADO.

Second, if the text is larger than the maximum text size possible in DADO†, the host breaks the text into blocks (which overlap in exactly the same way as the text partitions on the DADO). Each block is thus processed sequentially by DADO. The host translates the matching pointers reported by the DADO into the appropriate text block(s) during the report phase. Note this procedure places no limit on the maximum text allowable with the algorithm.

4.2 Match Algorithm. The match algorithm implemented in each PE is based on a well-known *dynamic programming* method as shown in Figure 2. The matrix $D_{i,j}$ stores the minimum number of differences between a substring of the search pattern $s_1, \ldots, s_i$ and any substring of the text ending with t_j. A straightforward implementation of this algorithm would require a memory space of *mn* and involve an expensive multiply operation on the DADO PE for the array address calculation. We modified this algorithm slightly for DADO as shown in Figure 3. The idea of the optimization is that a row of *D* is calculated only with reference to the previous row, thus memory space of 2*n* is enough for the calculation, provided each row is used alternately. Since the first array index is always 0 or 1, no multiplication is involved in the address calculation for the array $D[2][n]$. Aside from the reduced memory requirements as noted above, this results in a threefold improvement in compute time.

```
1. Initialization
        for j = 0 to $n$        $D_{0,j} = 0$
        for i = 1 to $m$        $D_{i,0} = i$
2. Computation
        for i = 1 to $m$
                for j = 1 to $n$        if $s_i = t_j$        $D_{i,j} = min(D_{i-1,j} + 1, D_{i,j-1} + 1, D_{i-1,j-1})$
                                        if $s_i \neq t_j$     $D_{i,j} = min(D_{i-1,j} + 1, D_{i,j-1} + 1, D_{i-1,j-1} + 1)$
3. Occurrence check
        for j = 1 to $n$        if $D_{m,j} \le k$        There is an occurrence ending at $t_j$.
```

Figure 2. *Basic Dynamic Programming Algorithm*

4.3 Time Complexity of the Algorithm. The time complexity of the algorithm shown in Figure 2 is *O(mn)*. From the data partition algorithm described in the previous section, it is clear that the time complexity for DADO is $O(\frac{mn}{p} + (m+k)m)$. The first term shows the improvement from data partitioning. The second term shows the penalty due to the overlapping data. Since the second term does not include a factor of $\frac{1}{p}$ and can be the same order as the first term, the number of differences and especially the search pattern size can significantly affect the search time.

The performance improvement (ratio of compute time) against sequential machines with the same algorithm is also estimated from the above time complexities as follows:

† the maximum text size in DADO is $p\,(2048-(m+k-1))$ bytes.

```
1. Initialization
        for $j = 0$ to $n$        $D_{0,j} = 0$
2. Computation
        for $i = 1$ to $m$
                if $i$ = odd number        $D_{1,0} = i$
                                           for $j = 1$ to $n$    if $s_i = t_j$     $D_{1,j} = min(D_{0,j} + 1, D_{1,j-1} + 1, D_{0,j-1})$
                                                                 if $s_i \neq t_j$  $D_{1,j} = min(D_{0,j} + 1, D_{1,j-1} + 1, D_{0,j-1} + 1)$
                if $i$ = even number       $D_{0,0} = i$
                                           for $j = 1$ to $n$    if $s_i = t_j$     $D_{0,j} = min(D_{1,j} + 1, D_{0,j-1} + 1, D_{1,j-1})$
                                                                 if $s_i \neq t_j$  $D_{0,j} = min(D_{1,j} + 1, D_{0,j-1} + 1, D_{1,j-1} + 1)$
3. Occurrence check
        if $m$ = odd number
                for $j = 1$ to $n$    if $D_{1,j} \leq k$    There is an occurrence ending at $t_j$.
        if $m$ = even number
                for $j = 1$ to $n$    if $D_{0,j} \leq k$    There is an occurrence ending at $t_j$.
```

Figure 3. *Match Algorithm on DADO*

$$\text{SEQUENTIAL MACHINE / DADO} = 1 \ / \ \mathrm{C}\left[\frac{1}{p} + \frac{m+k}{n}\right]$$

where C is a constant giving the ratio of the computing power of the sequential machine to that of a single PE for this algorithm.

Again, the first term of the equation is the improvement and the second term is the penalty. The second term also shows the upper bound on performance improvement for a certain size text. The performance will not improve beyond this value, regardless of the number of PEs.

4.4 Report phase. PEs that do not match are disabled. By use of the resolve function as described above, one of the enabled PEs is selected. The root PE collects matching pointers from the selected PE and sends them to the host. The selected PE is disabled, after it reports. This process is repeated until all enabled PEs are disabled.

5. Compute Time

We have measured the compute time for the match phase with various parameters. This section reviews the results and the procedures used. (Timing for the allocation and reporting phases is discussed in Section 5.3.) We have also compared the time against the VAX 11/750, running on 4.3 BSD UNIX† operating system. The same algorithm is used on both machines. Another algorithm on the VAX will be discussed in Section 6.

For the purposes of the experiments reported here, we used text files ranging in size from 1 Kb to 1 Mb. An arbitrary string was selected from the text and modified appropriately to suit its use as a search pattern.

† *UNIX is a trademark of AT&T Bell Laboratories.*

Timing on the DADO was accomplished using the timers built into the Intel 8751. Timer resolution is one microsecond. The elapsed time is that of the slowest PE, since the MIMD PE reads the timer after all PEs have finished a job. Timing on the VAX was accomplished using the *times* function provided by UNIX. The "user time" returned by the *times* function was used, that is, the total amount of time spent executing in user mode. Timer resolution is 1/60 second.

5.1 Compute time for small text portions. Let us start with text that is small enough to fit in one PE, since we are interested in comparing the computing power of a single PE. Table 1 shows the compute times for a text of 1 Kb with pattern sizes of 50, 150, and 250 bytes and using DADO machines of 1 PE, 2 PEs, and a maximum number of PEs that varies according to pattern size.†

TABLE 1. *Compute Time of DADO and VAX (1 Kb text)*

Pattern	*VAX*	*DADO*				
size	*11/750*	*1 PE*	*2 PEs*	*774 PEs*	*874 PEs*	*974 PEs*
50	*3.43*	*5.86 [1.00] (0.59)*	*3.07 [1.91] (1.11)*	-	-	*0.29 [20.21] (11.83)*
150	*10.23*	*17.55 [1.00] (0.58)*	*10.05 [1.75] (1.02)*	-	*2.58 [6.81] (3.97)*	-
250	*17.09*	*29.24 [1.00] (0.58)*	*18.16 [1.61] (0.94)*	*7.14 [4.09] (2.39)*	-	-

1. All times are in seconds. 2. Zero differences (i.e., exact matching). 3. [ ] : Computing speedup over single PE. 4. ( ) : Computing speedup over the VAX 11/750.

From Table 1, we see that the computing power of a single PE for this character-oriented operation is approximately 58% that of the VAX 11/750. This number was much larger than we expected since the VAX is a 32-bit machine with a rich instruction set, while the Intel 8751 is a tiny 8-bit microcontroller. We assume the reasons for this are: (a) simple operators such as *add, comp, mov* are used; and (b) most of the operands are characters. These factors make effective use of the 8-bit architecture.

5.2 Compute time for larger text. We measured the system for larger text size. Tables 2 and 3 (for 100 Kb and 1 Mb texts, respectively) show changes in compute time over a range of total differences. In each case, 1023 PEs are used.

We measured the relation between compute time and the number of PEs. Figure 4 shows computing speedup over 100 PEs. By increasing the number of PEs, the text is partitioned more finely. However, compute time in the overlapped area is constant for fixed values of m and k, and have larger weight when the text partitions are small. Therefore the computing speedup is diminished by these values. Using a 50-byte search pattern as an example, between 5% (100 PEs) and 33% (1023 PEs) of total compute time is spent for the calculation in the overlapping area. This number becomes large for larger pattern sizes: it occupies about 71% for a pattern of

† *In these cases, the text size is so small that, even with maximal partitioning as described in Section 4.1, the number of text partitions is less than the 1023 PEs physically available on the DADO2.*

250 bytes (1023 PEs), though DADO is nevertheless nearly 170 times faster than the VAX in that case.

Secondly, we compare the compute times of the DADO and the VAX. Figure 5 shows the experimental values for $\frac{compute\ time}{mn}$ with 1023 PEs. Note that compute time increases with pattern size as predicted by the time complexity model; the DADO compute time is proportional to the pattern size (more precisely, to $\frac{m+k}{n}$). The VAX compute time is almost constant at 68 microseconds.

TABLE 2. *Compute Time of DADO and VAX (100 Kb text)*

Pattern	*VAX*	*Number of Differences in DADO (1023 PEs)*					
size	*11/750*	*0*	*5*	*10*	*15*	*20*	*25*
50	*350.46*	*0.86* [1.00] (407.0)	*0.89* [0.97] (393.3)	*0.92* [0.94] (381.3)	*0.95* [0.91] (370.1)	*0.98* [0.88] (359.1)	*1.01* [0.86] (348.7)
150	*1056.53*	*4.28* [1.00] (246.7)	*4.37* [0.98] (241.9)	*4.45* [0.96] (237.3)	*4.54* [0.94] (232.7)	*4.63* [0.93] (228.4)	*4.71* [0.91] (224.2)
250	*1693.98*	*9.99* [1.00] (169.6)	*10.13* [0.99] (167.2)	*10.28* [0.97] (164.9)	*10.42* [0.96] (162.6)	*10.56* [0.95] (160.4)	*10.70* [0.93] (158.3)

1. All times are in seconds. 2. [ ] : Computing speedup over zero differences. 3. ( ) : Computing speedup over the VAX 11/750.

TABLE 3. *Compute Time of DADO and VAX (1 Mb text)*

Pattern	*VAX*	*Number of Differences in DADO (1023 PEs)*					
size	*11/750*	*0*	*5*	*10*	*15*	*20*	*25*
50	*3541.07*	*6.03* [1.00] (578.2)	*6.05* [1.00] (585.3)	*6.08* [0.99] (582.4)	*6.11* [0.99] (579.6)	*6.14* [0.98] (576.7)	*6.17* [0.98] (574.0)
150	*10498.11*	*19.75* [1.00] (531.5)	*19.84* [1.00] (529.1)	*19.92* [0.99] (527.0)	*20.01* [0.99] (524.6)	*20.09* [0.98] (522.6)	*20.18* [0.98] (520.2)
250	*17393.96*	*35.75* [1.00] (486.5)	*35.89* [1.00] (484.6)	*36.03* [0.99] (482.8)	*36.18* [0.99] (480.8)	*36.32* [0.98] (478.9)	*36.46* [0.98] (477.1)

1. All times are in seconds. 2. [ ] : Computing speedup over zero differences. 3. ( ) : Computing speedup over the VAX 11/750.

5.3 I/O time. This section discusses the time required for the allocation and reporting phases of our implementation.

5.3.1 Allocation phase. The time for the allocation phase is dominated by text loading time.

Prior to receiving text, each PE calculates a pointer to and size of the text partition. The root PE broadcasts the text in its entirety to all PEs. Each PE tracks its position in the broadcast text and stores into its local memory only that part of the text which is within its computation area; other text is ignored.

We estimate the data transfer rate to be 19 Kb/sec for the timing over a range of data sizes. It takes, for example, about 5 seconds to load a text of 100 Kb.

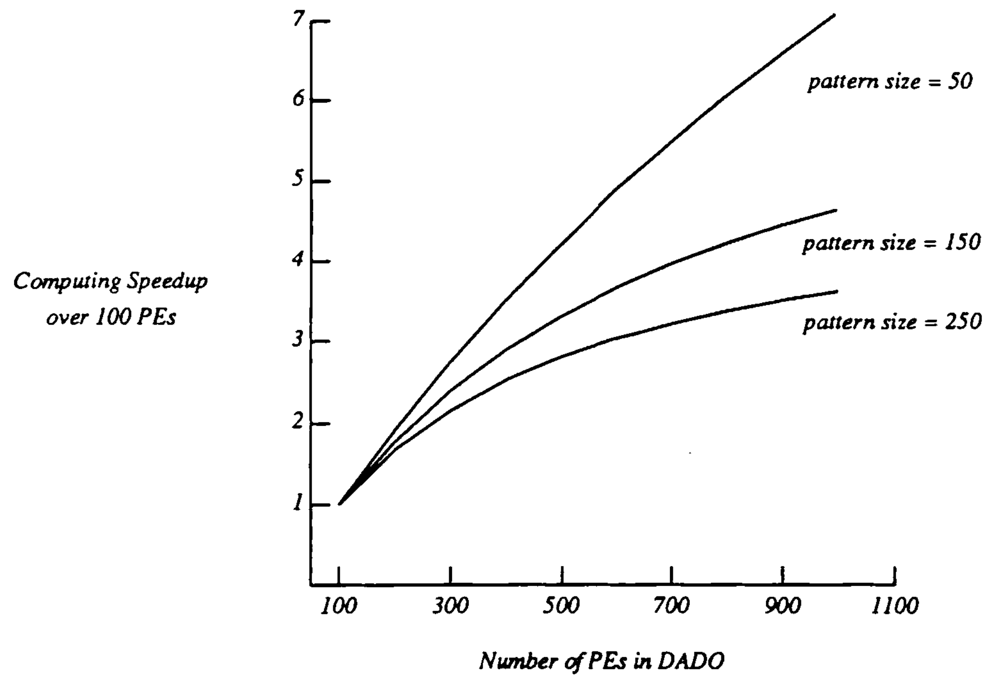

Figure 4. *Computing Speedup by number of PEs (100 Kb text, 0 differences)*

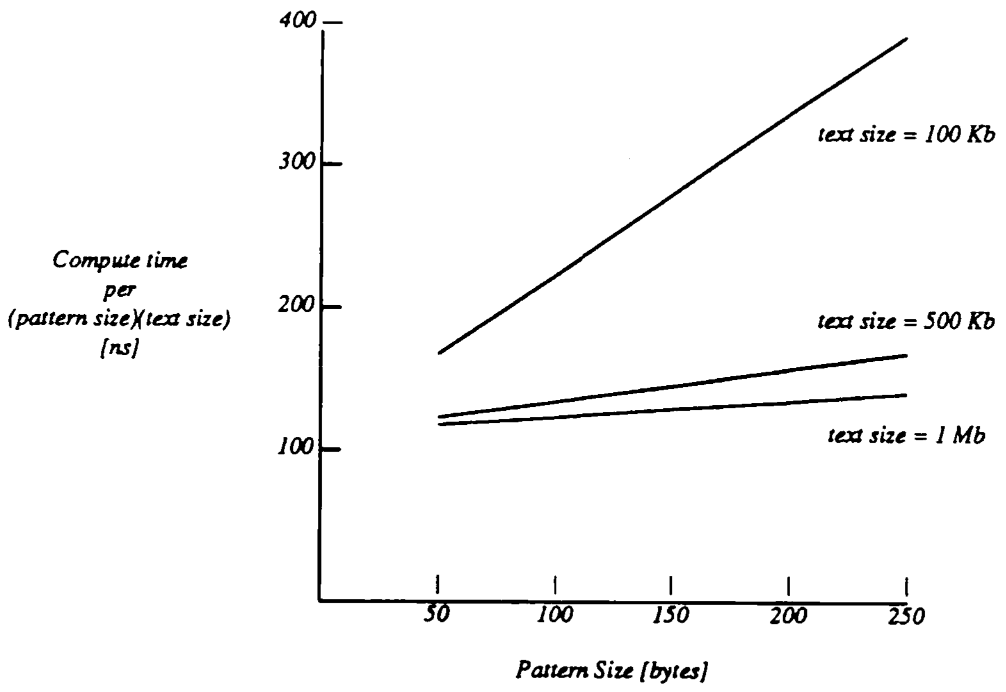

Figure 5. *Normalized Compute Time of DADO (1023 PEs)*

5.3.2 Reporting phase. In the report phase, only PEs that find one or more occurrences are enabled and take part in the operation. This phase has two steps.

Step 1: Each PE sends up the number of occurrences (2 bytes) to the root PE. The root PE counts the total number of occurrences (i.e., size of the data transfer in the next step; 4 bytes), and sends it to the host.

Step 2 : Each PE sends up the text partition pointer (4 bytes) and offset (2 bytes) for each occurrence. The root PE converts these values to absolute indices in the host's text and sends them to the host.

Assume that a total of s PEs found a total of r occurrences. Total timing t in this phase will be:

$$t = as + br + c$$

where a, b and c are constants. From our measurement, these constants may be replaced as follows:

$$t = 1.4s + 0.9r + 30 \ \textit{milliseconds}$$

For example, if 10 PEs found 2 occurrences (a total of 20 occurrences), it will take 62 milliseconds to send the result to the host.

Note that the number of PEs in the machine does not affect the I/O time, since the communication among the I/O chips is done in a pipelined manner, and because of the tree structure, 2^i PEs are in the same clock period where i is the depth from the root PE.

6. Alternative Match Algorithm

In this section, we discuss another algorithm for approximate string matching [8] [9] that displays noteworthy speed improvement under certain circumstances. This algorithm is shown in Figure 6. It uses the diagonals of the matrix D (see Figure 2) for its computation. In Figure 6, $L_{d,e}$ denotes the largest row i of D such that $D_{i,j} = e$ and $j - i = d$. *PREFIX* is a data structure where $PREFIX_{i,j}$ contains the length of the longest common prefix between the pattern suffix starting with s_i and the text suffix starting with t_j.

By preprocessing the text and the pattern, and constructing a suffix tree, *PREFIX* can be computed in constant time. Once such a data structure is constructed, the algorithm can be computed in the time complexity of $O(kn)$. The time complexity of the preprocessing step is estimated $O((n+m) \log min(\alpha, m))$, where α is the size of alphabet [9]. This algorithm is quite good when k is small.

The amount of memory space required for preprocessing may limit the implementation of this algorithm on the current DADO2. For the comparison between the VAX and DADO, we implemented the algorithm on the VAX as follows. *PREFIX* is constructed as a two-dimensional array, a straightforward implementation of the algorithm described above. Our implementation of *PREFIX* is easy to construct, but has a large time complexity compared to the above estimate. Thus, we discuss only the matrix compute time in Figure 6.

The normalized compute time (in this case, $\frac{compute\ time}{n\,(k+1)}$) *on the VAX is about 150* *microseconds. If a 1 Mb text is used, it will take 150 seconds for $k = 0$, while the compute time on the DADO is 36 seconds for $k = 0$ and $m = 250$, which means that DADO performs faster than the VAX. Note that compute time on the VAX is proportional to k. If $k = 25$, the VAX will take an hour, but the compute time on the DADO is almost the same as when $k = 0$.*

```
1. Initialization
        for $d = 0$ to $n + 1$                              $L_{d,-1} = -1$
        for $d = -(k + 1)$ to $-1$               $L_{d,|d-2|} = -\infty$; $L_{d,|d-1|} = |d-1|$

2. Computation and occurrence check
        for $e = 0$ to $k$
            for $d = -e$ to $n$
                $row = max(L_{d,e-1} + 1, L_{d-1,e-1}, L_{d+1,e-1} + 1)$
                $L_{d,e} = row + PREFIX_{row+1,row+1+d}$
                if $L_{d,e} \leq m$                 There is an occurrence ending at $t_{m+d}$.
```

Figure 6. *Alternative Match Algorithm*

7. Conclusion

The parallel computer DADO2 shows significant speedup over the VAX computer, although, in our implementation, compute time and its improvement against sequential computers are dependent on the searching parameters. For example, in a search for a 250-byte pattern in a 1 Mb text with 25 differences (10% of the search pattern), the DADO with 1023 PEs is over 400 times faster than the VAX 11/750 in compute time, and nearly 200 times faster when text loading time is included. It is worth pointing out that the timing results provided here are quite close to the complexity estimates given in Section 4.3.

One important application area of approximate string matching is string search for text retrieval systems. In previous research in parallel computers, Stanfill and Kahle reported free-text search on the Connection Machine† [10]. Stone analyzed their work and pointed out that "parallel query algorithms that do not use indexing may perform poorly relative to serial searches with indexing [11]." However once the problem is extended to a more general problem such as search with approximation, indexing is no longer adequate and parallel computers will allow favorable speedup as compared to sequential computers.

The application area is not limited to text retrieval systems, since *dynamic programming* techniques are central to a wide range of classical pattern matching tasks, as in speech recognition and genetic sequence matching [12].

The results reported here therefore demonstrate one method of accelerating these tasks on parallel hardware.

† *The Connection Machine is a registered trademark of Thinking Machines Corporation.*

Acknowledgements

We would like to thank Mark D. Lerner, Russell C. Mills, and Leland Woodbury for their help, encouragement and suggestions in various stages of this work.

References

[1] Patrick A. V. Hall and Geoff R. Dowling, "Approximate String Matching", Computing Surveys, Vol. 12, No. 4, Dec. 1980, pp. 381-402.

[2] L.R. Rabiner and S.E. Levinson, "Isolated and Connected Word Recognition-Theory and Selected Applications", IEEE Transactions on Communications, Vol. COM-29, No. 5, May 1981, pp. 621-659.

[3] Mark D. Lerner, Gerald Q. Maguire Jr., and Salvatore J. Stolfo, "An overview of the DADO parallel computer", Proc. National Computer Conference 1985, pp. 297-306.

[4] Salvatore J. Stolfo and Daniel P. Miranker, "The DADO Production System Machine", Journal of Parallel and Distributed Computing 3, 1986, pp. 269-296.

[5] Salvatore J. Stolfo, "Initial Performance of the DADO2 Prototype", Computer, Vol. 20, No. 1, Jan. 1987, pp. 75-83.

[6] Salvatore J. Stolfo, Daniel Miranker, and Mark D. Lerner, "PPL/M: The System Level Language for Programming the DADO Machine", tech. report, Dept. Computer Science, Columbia University, Feb. 1984. (CUCS-104-84)

[7] Michael van Biema, Mark D. Lerner, Gerald Q. Maguire Jr., and Salvatore J. Stolfo, "|| PSL: A Parallel Lisp for the DADO machine", tech. report, Dept. Computer Science, Columbia University, Feb. 1984. (CUCS-107-84)

[8] Gad M. Landau and Uzi Vishkin, "Introducing Efficient Parallelism into Approximate String Matching and A New Serial Algorithm", Proc. 18-th ACM STOC, 1986, pp. 220-230.

[9] Z. Galil and R. Giancarlo, "Data Structure and Algorithms for Approximate String Matching", Journal of Complexity, to appear.

[10] Craig Stanfill and Brewster Kahle, "Parallel Free-Text Search on the Connection Machine System", Comm. ACM, Vol. 29, No. 12, Dec. 1986, pp. 1229-1239.

[11] Harold S. Stone, "Parallel Querying of Large Database: A Case Study", Computer, Vol. 20, No. 10, Oct. 1987, pp. 11-21.

[12] David Sankoff and Joseph B. Kruskal (editors), "Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison", Addison-Wesley, Reading, USA, 1983.

Appendix A – Example

Here we present a sample session with one implementation of approximate string matching on the DADO2 machine; this implementation prints out all lines which contain approximate matches. In the presentation below, we have underlined the matching segments of each output line.

SEARCH PATTERN
"opendir"

TEXT STRING†

```
 1  "DESCRIPTION
 2       Opendir opens the directory named by filename and associates a directory
 3       stream with it.  Opendir returns a pointer to be used to identify the
 4       directory stream in subsequent operations.  The pointer NULL is returned
 5       if filename cannot be accessed, or if it cannot malloc(3) enough memory
 6       to hold the whole thing.
 7
 8       Readdir returns a pointer to the next directory entry.  It returns NULL
 9       upon reaching the end of the directory or detecting an invalid seekdir
10       operation.
11
12       Telldir returns the current location associated with the named directory
13       stream.
14
15       Seekdir sets the position of the next readdir operation on the directory
16       stream. The new position reverts to the one associated with the directory
17       stream when the tell dir operation was performed.  Values returned by
18       telldir are good only for the lifetime of the DIR pointer from which they
19       are derived.  If the directory is closed and then reopened, the telldir
20       value may be invalidated due to undetected directory compaction.  It is
21       safe to use a previous telldir value immediately after a call to opendir
22       and before any calls to readdir.
23
24       Rewinddir resets the position of the named directory stream to the
25       beginning of the directory.
26
27       Closedir closes the named directory stream and frees the structure
28       associated with the DIR pointer."
```

OUTPUT

```
k = 0 (exact match):
21:     safe to use a previous telldir value immediately after a call to opendir
k = 1:
 2:     Opendir opens the directory named by filename and associates a directory
 3:     stream with it.  Opendir returns a pointer to be used to identify the
21:     safe to use a previous telldir value immediately after a call to opendir
k = 2:
 2:     Opendir opens the directory named by filename and associates a directory
 3:     stream with it.  Opendir returns a pointer to be used to identify the
21:     safe to use a previous telldir value immediately after a call to opendir
27:     Closedir closes the named directory stream and frees the structure
```

† *The text is excerpted from "Unix Programmer's Manual: 4.3 BSD Virtual VAX-11 Version", University of California, Berkeley, California 94720, April, 1986.*