VLDC STRING MATCHING FOR ASSOCIATIVE COMPUTING AND MULTIPLE BROADCAST MESH

MARY C. ESENWEIN
EDS
4076 Youngstown-Warren Rd
Warren, OH 44484
330-373-7163
LNUSPKRD.mesenw01@eds.com

JOHNNIE W. BAKER
Kent State University
Dept. Mathematics and Computer Science
Kent, OH 44242
330-672-4004
jbaker@mcs.kent.edu

Abstract - This paper presents a new parallel algorithm for string matching with variable length "don't care" (VLDC). The initial computational model used is the associative computing model (ASC) enhanced with a linear network. ASC is a natural extension of the data parallel paradigm to a complete model for parallel computation. It supports massively parallelism through the use of data parallelism and constant time functions such as associative search and maximum value. It is also shown that the same algorithm is equally adaptable to the mesh with multiple broadcast. The algorithm has a run time of O(m) using O(n) processors, given a pattern of size m and a text of size n. The algorithm has the unique feature of permitting the identification of all match continuation points in the text after each "don't care" character.

Key Words - string matching, associative computing, ASC, parallel algorithms, data parallel programming

INTRODUCTION

String matching algorithms are among the more commonly analyzed algorithms in computer science. The extensive study of these algorithms is a result of their importance and general application: data validation, text editing, language translators, and DNA analysis. String matching is defined as finding all occurrences of a pattern string in another string called the text string. Formally, given a pattern string P of length m and a text string T of length n, where $m \leq n$, locate all positions i such that $T[i+j-1] = P[j]$, for all j where $1 \leq j \leq m$. The output is traditionally a boolean list of length n where list[i]= 1 if a match begins at T[i] and 0 elsewhere. An exact match means that the pattern string can be found intact within the text at least once. Optimally, sequential string matching algorithms have a run time of O(n).

Basic string matching algorithms look for all occurrences, including overlapping occurrences, of an exact match between a pattern and the text, but there are several variations to this problem. Generalized string matching, also referred to as dictionary matching, searches for occurrences of several patterns simultaneously. If any one of the patterns is found, a match is declared. In approximate matching algorithms, an attempt is made to convert the pattern (by deleting, inserting, or replacing characters) so that it will match a substring of text. A cost is applied to each activity performed against the pattern. If the cost of conversion is less than some user specified maximum, then a match is declared for that substring of text. A similar approach is called best match, where some match is always found, that being the sequence of text most like the pattern, thus requiring the least cost conversion. In multi-dimensional matching, patterns and text are expressed as multi-dimensional arrays. Any of these problems can be further complicated by the presence of a "don't care" character in the pattern. This is a special character outside of the problem alphabet that may be matched to any text character or substring of text characters, depending on the problem definition.

We narrow the scope of the subject in this paper to present a new algorithm for string matching with variable length "don't care" (VLDC) designed for the associative computing model, ASC. Additionally, we shall demonstrate that this same algorithm is also applicable on the mesh with multiple broadcast. By definition, VLDC string matching incorporates a wild card character, usually represented by '*', that automatically matches a text string of undetermined length. The use of a wild card in the pattern lends some flexibility to string matching algorithms by indicating that certain text characters are irrelevant to the matching process. Some VLDC algorithms may limit the number of characters '*' that may appear in the pattern or restrict '*' from appearing at the beginning or end of the pattern. The algorithm we present is not inhibited by either of these restrictions.

The best known run time for a sequential solution to VLDC string matching is O(mn/log n), presented by Myers [1]. Parallel VLDC algorithms are relatively rare. The best known time for a parallel algorithm is O(log n) using O(mn/log n) processors for the EREW PRAM, as described by Bertossi and Logi in [2]. The new algorithm presented here also has a cost of O(mn), but has the additional feature of being able to identify all continuation points of matching text after each "don't care" character. Our new algorithm is efficient in that it examines each pattern character once, comparing it only to those text characters that have the potential to lead to a successful match, and it finds all matches simultaneously.

ASC

The concept of associative memory is supported by certain parallel computers, called associative computers, by accessing objects in the local memory of each processor (PE) by content rather than by address. This is accomplished by broadcasting an item and having all active PEs search a specified field in their local memory for this data item, in parallel. A detailed account of the required features for an associative computer and how these features can be supported is given in [3].

The associative model, ASC, is a natural extension of the data parallel programming paradigm to a computational model that can simulate RAM. The ASC model supports a generalization of the associative style of computing that has been in use since the introduction of the STARAN associative computer by Goodyear in the early 1970's. A detailed description of this model is given in [4]. A high level language developed for ASC is described in [3] and has been installed on the STARAN, Goodyear/Loral/Martin-Marietta's ASPRO, the WaveTracer, and Thinking Machine's CM-2. In addition, an efficient simulator has been implemented on both PCs and workstations running UNIX [3, 9]. The ASC model provides an efficient and easy to program model for algorithms that utilize massively parallelism. As with data parallel programming, the programmer for the ASC model does not have to deal with the arduous programming chore of task allocation and, in fact, ASC programs are often shorter and simpler than well coded versions of their sequential counterparts. A massively parallel, general purpose computer that can directly support the ASC model is currently buildable. Also, this model is supportable on a wide variety of platforms. A brief description of this model appears in a recent textbook by Selim Akl [5]. A wide range of different type of algorithms and several large programs have been implemented using the ASC language including a parallel optimizing compiler for ASC, two rule-based inference engines, and an associative PROLOG interpreter [3, 9]. Some examples of algorithms for this model appear in [6, 7, 8, 9, 10]. We present here an overview of ASC and then use this model to present our VLDC string matching algorithm.

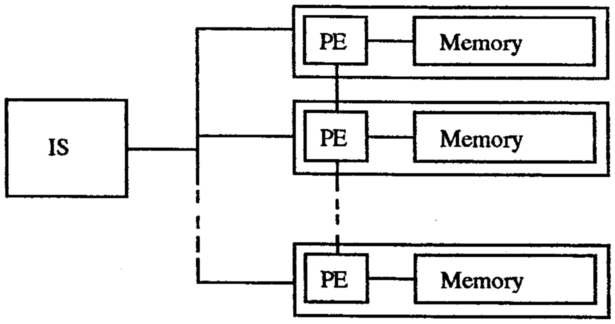

FIG. 1. ASC INSTRUCTION STREAM AND CELLULAR MEMORY

ASC consists of an array of cells, each comprised of a PE and its local memory. Cell memory holds variables used for data parallel operations. These cells are connected by a bus to a processor called the instruction stream (IS) which stores a copy of the program being executed and broadcasts program instructions to all active cells. The IS should not be confused with the concept of a host processor. Because the IS sends instructions to the PEs via a bus, the IS can be considered local to each PE. For our algorithm, only one IS is required, but the general ASC model allows multiple instruction streams. The number of ISs should be small relative to the number of PEs. While it is not a requirement of ASC, it is convenient to assume that variables and constants that need to be globally available to all cells are also stored in the memory of the IS and may be broadcast to all active cells. The IS also has the ability to read and store a value from a specific cell. The IS variables are considered scalar variables while the cell variables are considered parallel variables. To make a clear distinction between these two variable types in the algorithm, we have adopted the concept of adding a '$' suffix to the parallel variable identifiers.

In addition to data parallel execution, the ASC model supports constant time functions for associative searching and selection, logical operations, and maximum and minimum. Constant time searching permits the simultaneous examination of all active cells and the identification of all those that meet the search criteria. These identified cells are called responders and become the new set of active cells. By altering the criteria, different cells become responders. The IS has the ability to detect the presence of responders, access active cells in parallel or sequentially, and also to return to the set of cells which were active preceding the search. The constant time maximum (minimum) functions retrieve either the greatest (least) value of a cell variable or the index of the PE containing that value. The cells may also be connected to each other by means of a network such as a mesh, hypercube, or shuffle-exchange. The use of locality is important to avoid extraneous communications. Our algorithm assumes the use of a linear network. Figure 1 illustrates the ASC model.

The speed with which ASC can simulate PRAM gives a good indication of the power of ASC, but algorithms which run on ASC and simulate a PRAM algorithm may be less efficient than an inherently ASC algorithm. Let ASC(n, j) represent the ASC model of n PEs and j ISs, let PRAM(n, m) represent PRAM of n processors and m shared memories. ASC(n, 1) without a network can simulate a priority CRCW PRAM(n, m) in O(k) where k is the number if distinct memory locations accessed in this PRAM cycle and $k \leq \min\{n, m\}$. In general, ASC(n, j) without a network can simulate priority CRCW PRAM(n, m) in O(k/j) with high probability if the simulated PRAM memories are hashed among the ASC PEs. If ASC is enhanced with a network, then ASC can use both network techniques (at a cost based on the network used) and instruction streams to move data. An ASC(n, 1) machine with a network can simulate a priority CRCW PRAM(n, m) in min(O(k), O(route_net(n))), where route_net(n) is based on the fastest method to CR or CW n memory accesses for a given network. In general, ASC(n, j) can simulate combining CRCW PRAM(n, m) in min(O(k/j), route_net(n))) with a network with high probability if the simulated PRAM memories are hashed among the ASC PEs. Conversely, combining CRCW(n, m) can simulate ASC(n, j) in O(j) extra time and O(j/n) extra memory per PE. For additional details, see [11].

VLDC ALGORITHM

We assume that consecutive text characters are stored in separate but consecutive cells, starting with the second cell. The pattern is stored in and broadcast from the IS. In keeping with standard format, each '*' in the pattern represents an unspecified number of text characters that automatically match the pattern. A '*' may appear anywhere in the pattern, including the first and last positions, and multiple occurrences of '*' are permitted, however, two consecutive "don't care" characters are not allowed. Such use is unnecessary, anyway. Each '*' is counted as one pattern character when calculating the length of the pattern. The pattern is processed in reverse order, from P[m-1] to P[0].

Let each '*' separate the pattern into segments. Each segment will be treated essentially as if it were a separate pattern, with the condition that occurrences of one pattern segment will not be matched to text that lies beyond the last occurrence of a match to the previously tested pattern segment. Special handling is required when the pattern begins or ends with '*'.

We will now explain the functionality of the variables required by the algorithm. In addition to the text character variable, *text$*, each cell will contain a match indicator, *match$*, which is set to true if the algorithm determines that a match begins with the text character in that cell, and a counter, *counter$*, which will indicate the number of consecutive text characters that were successfully matched to the pattern prior to the current character. Each cell will also store an array, *segment$[k]*, for $0 \leq k < \lfloor n/2 \rfloor +1$, where *segment$[j]*, $0 \leq j \leq k$, is set to the length of the j'th pattern segment processed if the character *text$* for that cell begins a match to that segment. To accommodate all possible patterns, the size of the segment array must be $\lfloor n/2 \rfloor + 1$. This is dictated by the special case where m=n, n is even, and every other pattern character is '*', including the last character. Our example will only show the first three elements of *segment$[k]*, as that is all that is needed to illustrate this algorithm. The text is loaded with one character per PE for PE[2] through PE[n+1] where the PEs are indexed PE[1] through PE[n+1]. For practical application, the size of n should not be a concern. Each PE may store as much data as required and it is shown in [10] that an ASC model with may PEs can be simulated on an ASC model with fewer PEs if each PE of the smaller model contains the data of a fixed number of PEs from the larger model. The pattern is stored in an array in the IS and is processed in reverse order, from *P[m-1]* to *P[0]*. The IS also keeps a counter, *patt_counter*, which indicates the number of characters within the pattern that have been tested, the variable *maxcell*, which is the index of the highest cell to match the last pattern segment processed, and *patt_length*, the length of the pattern yet to be processed. All variables are initialized to zero except for *text$* , the pattern, *maxcell* (initial value n+2) and *patt_length* (initial value m). Each pattern character is examined once and, because of ASC capabilities, is matched simultaneously to all text characters that have the potential to lead to a successful match.

The algorithm is stated in figure 4. By following the example in figure 5, we can demonstrate the state of the associative memory after each pattern segment is processed for the problem of matching pattern AB*BB*A in text ABBBABBBABA. The segment array is reduced in size to show only the three elements needed for this particular example. The first pattern character processed is the final 'A'. The associative search activates all cells where *text$* = 'A', *counter$* = 0 and cell index < 13. This makes responders of cells 2, 6, 10, and 12. Each of these cells sends a message to its immediately preceding cell to set its *counter$* = 1. The variable *patt_counter* is then incremented to reflect the number of characters examined in the current pattern segment, as shown in figure 5(a).

When a '*' is encountered in the pattern, the second part of the while loop, statements 4 and 5, is executed. This code stores the length of the most recently matched pattern segment in the segment array of cells that begin a match to that segment. It also resets *maxcell* so that the next pattern segment will not look for responders beyond the last occurrence of a match with the most recent segment. Matching is terminated when it is determined that the pattern will not match the text, i.e. when a pattern segment fails to match a text substring, resulting in no responders to statement 5.

Looking at figure 5(a) again, after the final 'A' is processed, the '*' preceding it in the pattern is broadcast. The action initiated by the '*' is to activate those cells whose *counter$* = 1 (i.e. *patt_counter*) and to store the length of the last pattern segment (again, *patt_counter*) in *segment$[0]* of the cell following these responders (cells 2, 6, 10, and 12). This identifies these cells as beginning a match with the last pattern segment. Because cell 12 is the highest cell index to match that segment, *maxcell* is set to 12 so that the next pattern segment tested will not look beyond that point to find a match. Next, *patt_length* is decremented by the length of the pattern just processed, *patt_counter* is reset to zero for the next pattern segment, and the second 'B' of the middle pattern segment 'BB' is processed. The associative search for this character looks for a *text$* = 'B', a *counter$* = 0, and cell index < 12. Responders are cells 3, 4, 5, 7, 8, 9, and 11. These cells send a message to their preceding cells to set *counter$* = 1. The next pattern character is the first 'B' of the middle segment. The associative search is now a *text$* = 'B', a *counter$* = 1, and cell index < 12, resulting in cells 2, 3, 6, and 7 having *counter$* set to 2. The next pattern character is '*', so *segment$[1]* in cells 3, 4, 7 and 8 is set to 2 (the length of the middle segment as stored in *patt_counter*), *maxcell* is set to 8, and the process is repeated for the next segment, figures 5(b) and 5(c).

All text characters that successfully meet the associative search criteria for the first pattern character (i.e. the last character tested) are the text characters which begin a match to the whole pattern. Those cells (2 and 6) will set the match indicator, *match$*, to 1 (statement 6). Figure 5(d).

Special handling is required when the pattern begins or ends with '*'. When '*' is the first character of the pattern, then all text characters that precede the last occurrence of the first pattern segment are said to match the pattern. The processor that begins this last occurrence is captured in *maxcell* and all PE[i], for 0 < i < *maxcell*, may set their match indicator flag. Recall that the pattern is processed in reverse order so the beginning '*' is really the last character processed. All possible starting positions for all pattern segments are known at the time it is processed. The presence of a '*' at the end of the pattern means that any and all text characters beyond the first occurrence of a match to the last pattern segment will also match the pattern. In fact, all text characters would be considered a match to the end of the pattern and *segment$[0]* is set to indicate such. This action also handles the unlikely situation where '*' is the only pattern character.

Because all commands in ASC run in constant time, the run time for the algorithm is O(m) for O(n) processors. Other algorithms which solve VLDC problem can only identify the text positions where a start to the whole pattern match is found. While this practice is consistent with traditional output for the string matching problem in general, the unpredictable length of the text substrings represented by each '*' would indicate that it may be beneficial to be able to identify where the text pattern matching continues after each '*'. A unique feature of this algorithm is that it has captured all of the information necessary to find all continuation points of all matches following each '*'. The match to any pattern segment begins where *segment$[j]* is set to the length of the j'th segment, i.e. where *segment$[j]* > 0. The start of the continuation of the match to that segment is any text character whose *segment$[j-1]* is greater than zero and whose cell index is greater than or equal to the sum of the cell index of the current segment match plus the length of that current segment. Consider the match beginning in cell 2 of our example problem. The variable *segment$[2]* in cell 2 = 2, the length of the first pattern segment. Continuation points for this match are all cells where *segment$[1]* > 0 and whose cell index ≥ 4 (cells 4, 7, 8). Because *segment$[1]* = 2, any match that continues at cell 4 would continue in a cell whose cell index ≥ 6 and whose *segment$[0]* > 0 (cells 6, 10, 12). Similarly, any match that continues at cell 7 would continue at cells 10 and 12. By recursively following this rule, all continuation points for a match may be identified. Thus, for our example, these underscored matches are identifiable: ABBBABBBABA, ABBBABBBABA, ABBBABBBABA, ABBBABBBABA, ABBBABBBABA, ABBBABBBABA, ABBBABBBABA, ABBBABBBABA, ABBBABBBABA.

MESH WITH MULTIPLE BROADCAST

The mesh with multiple broadcast is an enhanced version of the m x n mesh network in that all PEs in a row/column are connected by a bus. See figure 2. All PEs are assumed to be identical and each is aware of its own coordinate (i,j) where $1 \leq i \leq n$ and $1 \leq j \leq m$. In addition to unit time communication between adjacent PEs, one PE may broadcast on its row/column bus to all other PEs listening on that bus in unit time. Other unit time functions that may be performed by all PEs simultaneously include: performing simple arithmetic of boolean operations; reading a message from the bus; writing a bit to the bus; and interpreting a bit string. This bit string may be interpreted as a sequence of all ones, all zeros, or a combination of ones and zeros. A more complete description of the mesh with multiple broadcast may be found in [12, 13].

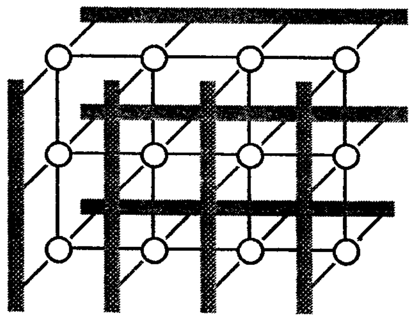

FIG. 2. MESH WITH MULTIPLE BROADCAST

To accommodate the new algorithm presented here, a simple 1 x (n+1) mesh with a bus connecting the processors is sufficient for a text of length n, where consecutive processors store consecutive text characters. See figure 3.

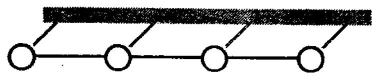

FIG. 3. 1 x n MESH WITH BUS

The functionality of the variables and the flow of the algorithm on this model is essentially the same as previously stated for the ASC model, except that the scalar variables *patt_counter*, *patt_length*, and *maxcell* must now be duplicated in each of the PEs and their updated values must be broadcast. Text characters are stored consecutively in PE[2] through PE[n+1]. The pattern will also be stored in and broadcast from PE[n+1]. Each processor PE[i], for $1 \leq i \leq n$ will compare its *text$* with a broadcast pattern character if *patt_counter$* in PE[i] = *counter$* and i < *maxcell$*, otherwise no comparison takes place. Then, PE[i] will transmit a value of 1 to PE[i-1] if PE[i] had a successful match, otherwise it will transmit 0. Each PE[i] will add the value it receives to its *counter$* and increment *patt_counter$* by one. A '*' is the pattern broadcast value will signal the end of a pattern segment. All PE[i] will transmit *patt_counter$* to PE[i+1] if PE[i].*counter$* = PE[i].*patt_counter$*, for $0 \leq i < n$. All other processors transmit 0. Each PE stores this transmitted value in *segment$[j]*. This means that a match to the j'th pattern segment begins at PE[i] if *segment$[j]* > 0 in PE[i]. It is necessary to reset *patt_counter$* and *counter$* to 0 before continuing with the next pattern character. It is also necessary to calculate the value of *maxcell$* and broadcast it to all processors.

The new value for *maxcell$* may be calculated and broadcast to all processors in constant time. It can be argued that the maximum function can be performed in constant time on the mesh with multiple broadcast. The time needed for this function would in fact be O(w) where w is the word size, a constant value for any given machine. In this case, the processor with the largest id number which begins a match to the most recent pattern segment is assigned to *maxcell$*. Each processor with *segment$[j]* > 0 sends its address, one bit at a time starting with the leftmost bit, to some accumulating processor. However, after each transmission, only those processors that sent a one bit will continue sending bits, the others drop out of contention. If no processor sends a one bit on any cycle, then all processors that sent the last zero bit send their next bit. The last processor to send a bit is assigned to *maxcell*.

On each iteration, *patt_length$* is decremented by *patt_counter$*, i.e. the length of the pattern yet to be processed is reduced by the size of the segment just processed. When *patt_length$* becomes zero, the whole pattern has been processed. Each processor PE[i] may set a flag indicating the start of a match to the whole pattern begins at the text character in PE[i] if *segment$[k]* > 0.

As with the ASC model, it is possible to terminate processing as soon as it can be determined that no match to a pattern segment is possible. As each pattern character is broadcast, each processor writes a bit to the bus indicating whether or not it matched the most recently broadcast pattern character. Then, PE[n+1] will interpret that sequence of bits. If the bit sequence contains all zeros, meaning no text character matched that pattern character, a complete match of the pattern is not possible, and the algorithm may be terminated.

All commands are executed in constant time, so the run time for this algorithm is O(m) using O(n) processors. Using the same logic as previously described, it is possible to identify of the matching continuation points after each occurrence of '*'.

CONCLUSION

We have presented a new algorithm for VLDC string matching. The algorithm was presented for the associative computing model, ASC, using one IS and enhanced with a linear network, and for the mesh with multiple broadcast for a 1 x (n+1) network. The algorithm has a run time of O(m) using O(n) processors, works for a general alphabet, and does not require any preprocessing or any knowledge of or periodicity in the pattern. Multiple occurrences of '*' are permitted and '*' may appear as the first and/or last character of the pattern. A unique feature of the algorithm is the ability to identify all match continuation points after each '*'.

ACKNOWLEDGMENT

The authors thank Darrell Ulm for his helpful comments on ASC/PRAM simulation.

REFERENCES

[1] Gene Myers, A Four Russians Algorithm for Regular Expression Pattern Matching, *J. Assoc. Comput. Mach.*, 39 (4), 1992, 430-448.

[2] Alan A. Bertossi, Filippo Logi, Parallel String Matching With Variable Length Don't Cares, *J.Parallel Distributed Comput.*, 22 (2), 1994, 229-234.

[3] Jerry Potter, *Associative Computing: A Programming Paradigm for Massively Parallel Computers* (New York; Plenum Press, 1992).

[4] Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, Chandra Asthagiri, ASC: An Associative-Computing Paradigm, *Computer*, 27(11), 1994, 19-25.

[5] Selim G. Akl, *Parallel Computation: Models and Methods* (New Jersey; Prentice Hall, 1997).

[6] Maher M. Atwah, Johnnie W. Baker, Selim Akl, An Associative Implementation Of Classical Convex Hull Algorithms, *Proceedings of the Eighth IASTED International Conference on Parallel and Distributed Computing and Systems*, 1996, 435-438.

[7] Maher M. Atwah, Johnnie W. Baker, Selim Akl, An Associative Implementation of Graham's Convex Hull Algorithm, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 1995, 273-276.

[8] Mary C. Esenwein, *Parallel String Matching Algorithms Using Associative Computing and Mesh with Multiple Broadcast*, Masters Thesis, Kent State University, 1995.

[9] Darrell R. Ulm, Johnnie W. Baker, Solving a 2D Knapsack Problem on an Associative Computer Augmented with a Linear Network, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1996, 29-32.

[10] Darrell R. Ulm, Johnnie W. Baker, Virtual Parallelism by Self Simulation of the Multiple Instruction Stream Associative Model, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1996, 1421-1430.

[11] Darrell R. Ulm, Johnnie W. Baker, The Power of the Associative Model Compared to PRAM, Technical Report, Kent State University.

[12] D. Bhagavathi, P, J, Looges, S. Olariu, J. L. Schwing, Selection on Rectangular Meshes with Multiple Broadcasting, *Bit*, 33, 1993, 7-14.

[13] D. Bhagavathi, S. Olariu, W. Shen, L, Wilson, A Time-Optimal Multiple Search Algorithm on Enhanced Meshes, with Applications, *J. Parallel Distributed Comput.*, 22, 1994, 113-120.

```
     int patt_length = m;
     int maxcell = n + 2;

      /* Special handling for '*' at end of pattern */
1.   if (pattern[m-1] == '*')
     {  Responders are cell index > 1;
           Responders set segment$[0] = 1;
        patt_counter = 1;
        k = 1;  /* reset initial segment index */
     }

2.   while ((patt_length -= patt_counter) >  0
               &&  maxcell > 0)
      {  patt_counter = 0;
         for (i = patt_length - 1;  i >=0  && pattern[i] != '*';  i–)
3.       {    Responders are
                  text$ == pattern[i]
                  and counter$ == patt_counter
                  and cell index < maxcell;
              Responders add 1 to counter$
                  and store result in counter$ of  preceding cell;
              patt_counter++;
         }

4.       Responders are counter$ == patt_counter;
              Responders set segment$[k] = patt_counter
                in next cell;

5.       Responders are segment$[k] > 0;
            maxcell = maximum cell index value of Responders
         else if no Responders,
            maxcell = 0;

         All cells become Responders and set counter$ = 0;
         patt_counter++;   k++;
     }

       /*  When pattern has been processed:  */
6.     Responders are segment$[–k] > 0;
          Responders set match$ = 1;

       /* Special handling for '*' at start of pattern  */
7.   if (pattern[0] == '*')
     {  Responders are cell index < maxcell
                        and cell index > 1; /* skip first cell  */
        Responders set match$ = 1;
     }
```

FIG. 4. VLDC ALGORITHM USING ASC

	T$	M$	C$	S0$	S1$	S2$
1	@	0	1	0	0	0
2	A	0	0	1	0	0
3	B	0	0	0	0	0
4	B	0	0	0	0	0
5	B	0	1	0	0	0
6	A	0	0	1	0	0
7	B	0	0	0	0	0
8	B	0	0	0	0	0
9	B	0	1	0	0	0
10	A	0	0	1	0	0
11	B	0	1	0	0	0
12	A	0	0	1	0	0

(a)

After third pattern segment

	T$	M$	C$	S0$	S1$	S2$
1	@	0	0	0	0	0
2	A	0	2	1	0	0
3	B	0	2	0	2	0
4	B	0	1	0	2	0
5	B	0	0	0	0	0
6	A	0	2	1	0	0
7	B	0	2	0	2	0
8	B	0	1	0	2	0
9	B	0	0	0	0	0
10	A	0	1	1	0	0
11	B	0	0	0	0	0
12	A	0	0	1	0	0

(b)

After second pattern segment

	T$	M$	C$	S0$	S1$	S2$
1	@	0	2	0	0	0
2	A	0	1	1	0	2
3	B	0	1	0	2	0
4	B	0	1	0	2	0
5	B	0	2	0	0	0
6	A	0	1	1	0	2
7	B	0	0	0	2	0
8	B	0	0	0	2	0
9	B	0	0	0	0	0
10	A	0	0	1	0	0
11	B	0	0	0	0	0
12	A	0	0	1	0	0

(c)

After first pattern segment

	T$	M$	C$	S0$	S1$	S2$
1	@	0	0	0	0	0
2	A	1	0	1	0	2
3	B	0	0	0	2	0
4	B	0	0	0	2	0
5	B	0	0	0	0	0
6	A	1	0	1	0	2
7	B	0	0	0	2	0
8	B	0	0	0	2	0
9	B	0	0	0	0	0
10	A	0	0	1	0	0
11	B	0	0	0	0	0
12	A	0	0	1	0	0

(d)

Final state

FIG. 5. STORED DATA VALUES FOR BOTH MODELS

VLDC MATCH FOR PATTERN AB*BB*A IN TEXT ABBBABBBABA