Order Number 8801732

String matching: A comparative study of algorithms, and its relation to problems of parallel and distributed computing

Lucci, Stephen, Ph.D.

City University of New York, 1987

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

STRING MATCHING

A Comparative Study of Algorithms, and its Relation to Problems of Parallel and Distributed Computing

by

STEPHEN LUCCI

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1987

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

May 12, 1987
Date

Michael Anshel
Chair of Examining Committee

May 12, 1987
Date

L S Beckman
Executive Officer

Professor George Ross

Professor Valentin Turchin

Professor Jacob Rootenberg

Professor Stanley Habib

Professor Chaim Ziegler

Supervisory Committee

The City University of New York

Abstract

STRING MATCHING

A Comparative Study of Algorithms, and its Relation to Problems of Parallel and Distributed Computing

by

Stephen Lucci

Adviser: Professor Michael Anshel

Our purpose in this work is to provide a comparative study of string matching algorithms and their relationship to problems in parallel and distributed computing. We begin our discussion in chapters one and two with the Knuth–Morris–Pratt and Boyer–Moore algorithms. These optimal serial approaches to string matching provide a reference point against which their parallel counterparts in chapters three through five may be judged. Galil's parallel algorithm in chapter three searches for prefixes of the pattern of increasing length which occur within the text string. Vishkin's Algorithm, described in chapter four employs the concept of duels between various text positions to obtain a sparse set of indices within the text in which the pattern might begin. In chapter five the Rabin–Karp Algorithm is outlined. This approach uses hashing functions called fingerprints, rather than character comparisons, as its basic operation. It is a serial algorithm

which unlike other serial approaches *is* readily parallelizable.

In chapter six we outline an architecture for a parallel processor for Galil's algorithm. The "parallel string processor" (PSP) is an SIMD computer with 2^{16} PEs and a perfect shuffle interconnection network. In the process of specifying our design we consider sorting algorithms, connection networks, and various paradigms for data broadcasting. Hopefully, this work foreshadows the rapidly unfolding technological breakthroughs that will serve as a bridge to fifth generation systems.

Preface

This thesis presents the basic concepts, techniques and methods underlying string matching, both in its classical sequential form and in very recent formulations in the context of parallel and distributed computing.

The central results culminate in a Parallel String Processor (PSP), a synthesis of a very high level parallel algorithm arising from the work of Z. Galil and a broadcasting scheme based on the perfect shuffle interconnection network which was first popularized by H. Stone.

We briefly describe the parallel string matching algorithm. The pattern X of length n is assumed to be present in the first n Processing Elements, of our PSP, with one character residing in each PE. The text which is assumed to be twice the pattern length is stored similarly within the first 2n PEs. The algorithm consists of $\lceil \log n \rceil$ stages. In stage one all prefixes of the pattern of length two, denoted by $X^{(1)}$ are sought. More generally in stage $i+1$, the algorithm attempts to uncover all prefixes in the input string of length 2^{i+1}. These prefixes, denoted by $X^{(i+1)}$ have the format $X^{(i)}Y^{(i)}$ where $X^{(i)}$ is just a pattern prefix of length 2^i. In stage $i+1$, it is necessary to verify whether occurrences of $X^{(i)}$ also correspond to instances of $X^{(i+1)}$. Naturally, only the $Y^{(i)}$ suffix need be checked. Our algorithm

employs a broadcasting mechanism, which in stage $i+1$, efficiently distributes the pattern characters comprising the $Y^{(i)}$ substring, precisely to those Processing Elements which contain the appropriate segments of the input string. This search for pattern prefixes incidentally, transpires within the pattern portion of the input as well – a strategy which enables the algorithm to uncover pattern periodicity. Periodic pattern prefixes are handled somewhat differently so as to not overburden the processor melange. After $\lceil \log n \rceil$ stages, occurrences of the pattern X are written into local memories from whence they may be output onto a secondary storage medium.

Although the Parallel String Processor is a gedanken experiment, we feel it has sufficient substance for others to employ as a template for testing their own ideas.

We cite some key components of the PSP design. 2^{16} PEs comprise the processing ensemble of our SIMD computer. Each Processing Element contains a pattern register and a text register to hold a single character from the input. Results of various comparisons within the pattern and text strings are recorded in two one-bit registers – a pattern switch and a text switch. An end of pattern and end of text register are employed for delimiter purposes. Two additional registers are enlisted for holding, in stage $i+1$, the components of the $Y^{(i)}$ substring. Also, four registers per processor are required for local communication. The first two of which — the pattern local bulletin board and the other pattern local bulletin board

are useful in detecting periodicity within the pattern, whereas the text local bulletin board and other text local bulletin board aid in detecting matches with X within the text proper. The adjective "local" refers to the fact that processors work in blocks to extend the length of detected prefixes of X within the input. In stage $i+1$, block size is 2^{i-1}. That is, each set of 2^{i-1} processors work in unison to verify instances of $Y^{(i)}$ where appropriate.

A global bulletin board is employed in the periodic phase of the algorithm. This register will contain the period size, P and the length of the periodic substring, L. Its contents are broadcast to all processors in the collection.

The processors in the PSP communicate via a perfect shuffle interconnection network. We provide techniques which permit our machine to efficiently distribute $Y^{(i)}$ substrings. The network is also employed to broadcast local and global bulletin board contents.

We hope that this overview will help the reader in traversing the landscape of this thesis.

Acknowledgements

Obtaining a Ph.D. is an emotional as well as an intellectual obstacle course. I would like to take this opportunity to thank the many people who have provided help in both arenas. First and foremost is Prof. Anshel who has been more like a friend than a mentor over the long and arduous path which culminated in this work. His infinite patience (well ... almost) helped me to sustain the necessary vision and perseverence to complete this study. Whenever I'd become discouraged, he was always there to remind me that science progresses more often in orderly steps than in giant leaps and bounds. I would also like to thank Ziva Anshel for her support and encouragement. Her strong espresso over the years has also been appreciated.

I would also like to express my gratitude to the members of my Examining Committee. Prof. Ross who has also been more like a friend than a chairman and who was always willing to schedule an additional Formal Language or Computability section when I was trying to learn some of the theory underlying our subject. Prof. Rootenberg of Queens College who provided career guidance as well as a solid education in hardware. He is also to be thanked for supplying the "gentle nudging" that led me to in-

corporate parallelism into my thesis. Prof. Turchin has been supportive throughout my graduate student years. His careful attention to detail at my second exam and his support during the final month before my defense are cheerfully acknowledged. I would also like to thank Prof. Beckman, executive officer of the Ph.D. Program for overseeing all the details entailed in scheduling a defense. Professors Habib and Zeigler are also to be thanked for agreeing to serve on my committee on relatively short notice. Dr. Gewirtz of Bell Laboratories was able to participate in my second exam and his time was certainly appreciated. Prof. Stebe, of the City College Math department was to have served on my committee but became ill at the last minute. He is certainly wished a speedy recovery.

I would be remiss if I did not take this opportunity to thank the entire faculty of the City College Computer Science department for the excellent education I have obtained. Special thanks are extended to Prof. Burr for the confidence he has expressed in my ability. And also for the many stimulating conversations in the corridors of Steinman Hall. Dr. Lidor, now of Bell Laboratories was a strong guiding force in my early years at City College. He always gave freely of his time and of himself. His support has not been forgotten.

I have had the great fortune of having many fine students at City College in my classroom over the years. And as has often been said, I feel that I have learnt more from them than the other way round. Several students

contributed to the readability of this work. Yoo Park, James Vita, Ti Jin and Kathleen Crowley have helped to proofread various sections of this manuscript. Jia Zhong Ding, a visiting scholar from China deserves special mention for finding typing and logical error "pins" in this "haystack of a manuscript".

Kathleen, mentioned above also provided some much needed last minute assistance, as did Robert Goldman, Dwen-Ren Tsai, Herman Weinstein, Athanasios Glavas, Marc Aronson and John Anspach. Also to be thanked are Minna Fox and Regina Holloman of the Computer Science department who organized a party on my behalf which helped to hasten my post defense recuperative phase.

Transforming a dissertation from scribblings on yellow scratch paper to a finished manuscript can be a long and horrifying nightmare. I was fortunate to have several people who gave freely of their time and talent in this regard. I would like to thank Jian Chen, Chia-Wen and Yuh-Yann Wu, Eva Chang and Jane Hsu. Jane, a good friend, also provided much needed encouragement during a dry spell which preceded the eventual completion of this work. She should also be thanked for converting the "spaghetti bowl" of wiring diagrams into figure 6.9 and also for squeezing so much data legibly into the balanced tree in figure 4.3. Barbara Tai provided technical assistance which facilitated the distribution of this manuscript to committee members and the library as well. Also, Prof. Vulis provided some

much needed and appreciated assistance in "taming" the CUNY computer system.

During most of the time that I attended graduate school, I worked on the night shift. Walter Murawinski was a friend who supplied encouragement and guidance during a time in my career when completing a Ph.D. degree seemed light years away. It has been a while since then, but his friendship is still remembered.

I would also like to acknowledge the kinship of Dipak Basu and Gordon Bassen, two people who have traveled over the same foreboding terrain. Memories of nights spent poring over unending stacks of text books accompanied by stale donuts, pungent coffee and these two friends remain indelibly etched on my mind.

Finally, I would like to thank my sister Rosemary who helped in the preparation of an earlier version of this work. And my parents who always encouraged my educational endeavors.

Contents

Figures and Tables

CHAPTER 1

KNUTH MORRIS PRATT ALGORITHM

§1.1 *Introduction*

String matching is a problem which arises naturally in many areas of Computer Science. Often, one wishes to find the occurrence of one string (the pattern) within a second, usually much longer string (the text). Several examples follow.

Compilation is the process of translating programs written in a high-level language (a source program) into machine code (an object program). Here, the high-level language program is the text. Lexical analysis is the first stage in compilation. At this stage, characters must be grouped into logical units called tokens. These logical units (identifiers, keywords such as IF, PERFORM, etc., or signs, e.g. + or −) are the patterns which the lexical analyzer must search for. As an example, in ALGOL 60 any recognizer for real numbers would have to label each of $dd^*.dd, .dd^*$ or dd^* as a decimal number; here d denotes a decimal digit. In practice, a finite

state accepter is built for such recognition tasks. A recognizer for ALGOL 60 numbers may be found in [*BACK*79]. Lexical analysis consumes a major portion of the time required for compilation. Hence, efficiency is a primary concern. A nice introduction to building efficient lexical analyzers is [*JOHN*68].

Text editors are often confronted with the problem of searching through a program listing (the text) to find the first, or all occurrences of a pattern string. Often the pattern is a variable name which is to be modified or deleted. Searching through a bibliography, for example, to find all listings with the words "string matching" in them; or through a large employee database to locate all single employees without health insurance, can be seen as other instances of the basic string matching problem.

In some computer graphics systems, pictures are represented as binary strings. Some textbooks are produced on such systems. Here, proper layout of a manuscript, will depend on various string matching problems. Notice, that in these latter examples, the problem has been extended to two dimensions. In a similar vein, edge detection in digital pictures, or the optimal layout of VLSI circuits on silicon (or galium arsenide) can be aided by efficient string matching algorithms.

String matching also plays a role in speech recognition. This problem is complicated by the varied speech habits of different speakers, viz. some people speak slower than others. Nucleic acid identification, gas chromatog-

raphy and dendrochronology, the study of dating based on tree rings are other applications. String matching is also useful in string correction, the automatic correction of human errors at the input stage, e.g. at a computer terminal. The reader is referred to a survey paper [$KRUSK83$] for a discussion of these and many other interesting applications.

At this point, we develop an algorithm to solve the basic string matching problem. As input, we have a pattern, whose length is patlen, and a text with length textlen. We use i as an index to the pattern string and j serves as a pointer within the text string. The output of our algorithm should be that location within the text string at which the first occurrence of pattern within text begins. If the pattern does not occur within the text, our algorithm should return the value $j = textlen + 1$.

It might be helpful to first consider an example. We take $V = \{a, b\}$ as our alphabet. Let the pattern be *abaab* and our text *ababaab*. A straightforward solution would start at the beginning of both strings, comparing characters one at a time until we find a match or j equals $textlen + 1$.

```
PATTERN :    a b a a b
             ↕ ↕ ↕ |
TEXT :       a b a b a a b
```

We let vertical arrows denote matched characters. It isn't until the fourth character in both the pattern and text that a mismatch is detected. We slide the pattern one position to the right, relative to the text and begin the matching process again. Actually, there is no sliding going on, only indices begin reset.

```
PATTERN :      a b a a b
                   |
TEXT :       a b a b a a b
```

A mismatch occurs immediately. $Pattern(1) \neq text(2)$ i.e., $a \neq b$. We slide pattern to the right once more.

```
PATTERN :      a b a a b
                 ↕ ↕ ↕ ↕ ↕
TEXT :       a b a b a a b
```

A match is found. The algorithm returns the value $j = 3$. This straightforward algorithm is given below. $P(i)$ and $t(j)$ refer to pattern and text characters, respectively.

Straightforward Approach

```
i ← 1; j ← 1
while i ≤ patlen and j ≤ textlen do
  begin
    if p(i) = t(j)
      i ← i + 1; j ← j + 1
    else j ← j − i + 2; i ← 1
  if  i = patlen + 1 then  j ← j − i + 1
return j
```

The time complexity of this algorithm depends upon the number of character comparisons made. If the pattern occurs at the beginning of the text, or if the first pattern character does not occur in the text, this straightforward algorithm works well. The worst case will occur when we obtain many partial matches, but continually fail to obtain a total match. Consider, for example the pattern $a^m b$ in the text a^n (where $n > m$). We repeatedly find m successful matches, until the "b" in the $(m+1)^{st}$ position of pattern always fails to match the corresponding text character. The worst case behavior of this algorithm is quadratic in the length of the pattern and text, i.e., Algorithm Straightforward is $O(mn)$.

We just discovered that a straightforward algorithm for pattern matching has time complexity $O(mn)$. When confronted with a method of solution to a particular problem, a Computer Scientist will invariably ask: "Can we do better?". If the problem were searching, one would quickly become disheartened with an $O(n)$ approach, e.g., sequential search. Instead, one might wish to employ binary search, with logarithmic complexity *if* the files to be searched are sorted, or even hashing with $O(c)$ average time complexity, if one is *not* concerned with worst case behavior. But why might one suspect that a "better" solution to the string matching problem is possible? To answer this question, we make a slight digression.

A 2DPDA (Two-way Deterministic Pushdown Automata) may be thought of as a two-tape Turing machine where one of the tapes is used as a pushdown store (see figure 2.1 below).

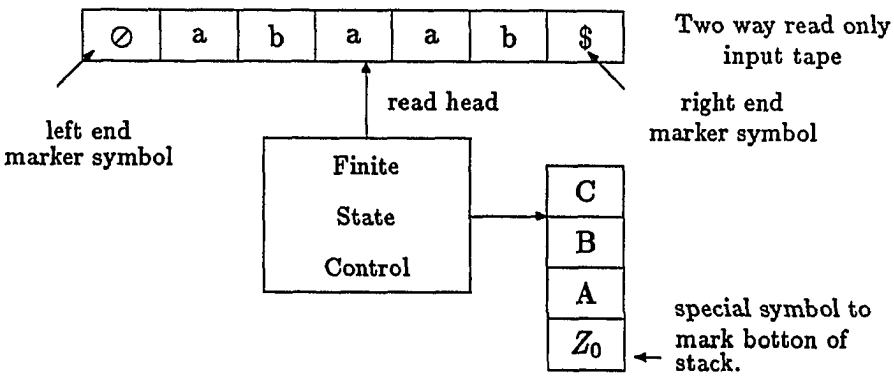


Figure 1.1: A two-way deterministic pushdown automata.

A 2DPDA has a read only input tape. An input string consisting of symbols drawn from an input alphabet I, is placed, one symbol per square between the two endmarkers (see figure 1.1). The input tape head reads one symbol at a time. In one move, the input tape head can shift left or right one square, or remain stationary. The pushdown list holds symbols from a pushdown alphabet. The finite state control is in a state drawn from S, its finite state set. The action of the machine is specified by its next-move function σ. For each input symbol scanned, top stack symbol viewed, and state that the machine is in, σ must specify what the next state is to be, whether the input head will move either left or right, or remain stationary and whether the stack contents should be altered (i.e. push a new symbol, pop the topmost symbol or do nothing to the stack).

String matching may be viewed in the context of a language recognition problem. Let L be the language $\{xcy|\ x,y$ are in $I^*, c \notin I$, and y is a substring of $x\}$. "Does the pattern y occur in the text x ?" is equivalent to determining whether the string $xcy \in L$.

Now comes the point of our digression. According to a theorem of Cook's, any language recognized by a 2DPDA, in whatever time, can be recognized on a Random Access Machine in *linear time*.

A surface configuration of a 2DPDA consists of a state that the control is in, the position of the input head, and the *topmost* symbol on the pushdown stack. An instantaneous description of such a machine would

also include the state and input head position, and the *entire* contents of the pushdown stack. In describing the moves of a 2DPDA, an exponential number of surface configurations would be required. The simulation depends upon following the behavior of a 2DPDA by using surface configurations, rather than instantaneous descriptions to describe moves of the machine. The interested reader may refer to [*AHO*74] for the details of this simulation. Knuth laboriously followed through the steps of the simulation before discovering a linear time pattern matching algorithm. He was dismayed to learn that Morris, working independently had discovered the same algorithm *without* using Cook's theorem.

§1.2 KMP algorithm

The essential idea behind the Knuth-Morris-Pratt (*KMP*) algorithm is this: When a mismatch occurs at position i in the pattern, there is no need to re-read the last $i-1$ characters of the text, they are the same as the first $i-1$ characters of the pattern ! What is needed, is a pre-processing of the pattern to take advantage of this knowledge. A flowchart needs to be constructed which indicates where to re-commence our search once a mismatch has been detected. Building a traditional finite state accepter, i.e. fsa (see [*HOP*79] or [*DEN*78] for definitions) would be cumbersome. An fsa for the pattern *abaab* is shown in figure 1.2.

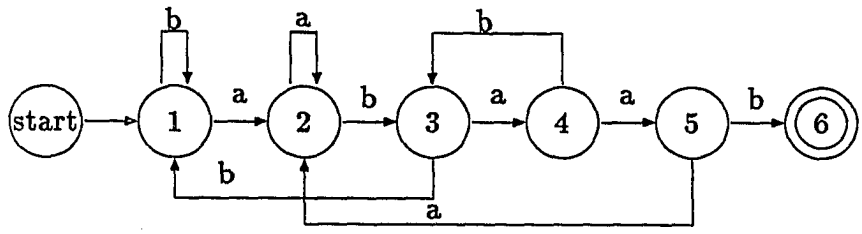


Figure 1.2: An fsa for the pattern *abaab*.

For each character read, a new starting position within the pattern must be computed. For even moderately long patterns and typical size alphabets (e.g. 26 characters as in English), building and efficiently using an fsa are problems. The space required by such a machine would be excessive. However, Tarjan and Yao [*TAR*79] describe an algorithm which would require only linear space (linear in the cardinality of the state set) and logarithmic access time. Their technique maintains a trie structure and employs double displacements to compress tables. Still, the time required to build and use such a structure make the fsa approach uninviting. More recently Galil

and Seiferas [*GALIL83a*] have developed an implementation for the string matching problem requiring linear time and a constant number of local storage locations. Their algorithm may be run on a six-head finite automation or even as a Fortran subroutine (if one is perhaps less theoretically inclined).

The KMP algorithm attempts to eliminate the drawbacks of the fsa approach by containing just two types of arrows, those to be followed on a match – success links, and those to follow when a mismatch occurs – failure links. Filling in the success links is easy (see figure 1.3 below).

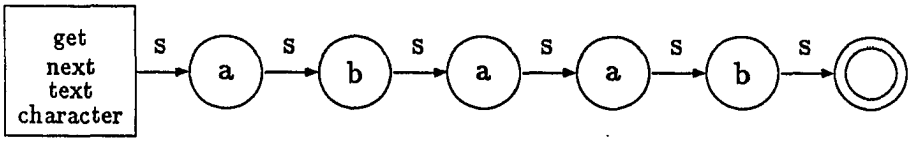


Figure 1.3: Filling in success links in the KMP algorithm.

Note that character labels occur on the nodes, and not on the arrows as in figure 1.2. A new text character is read only when a success arrow is followed. If a failure link is traversed, the same text character is compared

with a different pattern character. In this respect, failure transitions are similar to ϵ-transitions in an nfa (non-deterministic finite state accepter), i.e., no input is "consumed" during state changes. We must still fill in the failure links in this KMP machine. If a mismatch occurs at position i in the pattern, we require that $f(i)$, the failure link at position i, point to that location k which maximizes our use of prior information (i.e. the successfully matched characters which preceded the mismatch). Namely, $f(i)$ should point to the largest suffix of the text string read so far which equals a prefix of the pattern string, i.e., $t_{j-k+1} \ldots t_{j-1} t_j$ equals $p_1 p_2 \cdots p_k$ where $k < i, k$ maximal.

Consider the pattern *abaab* again and table 1.1 below.

PATTERN:	a b a a b
INDEX i:	1 2 3 4 5
f(i) :	0 0 1 1 2

Table 1.1: Failure function in the KMP algorithm for the pattern *abaab*.

We observe in table 1.1 that $f(3) = 1$. Let us suppose that j characters from the text string have been read, i.e. $t_1 t_2 \ldots t_j$. Further, suppose that our pattern matching machine is in state 3. Therefore, the last three characters from the text string match the first three characters of the pattern, i.e.

$t_{j-2}t_{j-1}t_j = p_1p_2p_3 = aba$. The $(j+1)^{st}$ text character is now compared with the $(i+1)^{st}$ pattern character (here i has a value of 3). We are asking if $t_{j+1} = p_4$, i.e. is the $(j+1)^{st}$ character of text an "a"? Suppose not. Then we have the following picture. Question marks stand for characters we are not interested in, i.e. have already been "processed".

$$\begin{array}{lccccccc} (*) & t_1 & t_2 & \ldots & t_{j-2} & t_{j-1} & t_j & t_{j+1} \\ & ? & ? & \ldots & a & b & a & x \\ & & & & p_1 & p_2 & p_3 & p_4 \end{array}$$

We have that $x \neq$ "a". We slide our pattern to the right with respect to the text. Our slide is of $i - f(i)$, or $3 - 1 = 2$ positions.

We obtain the situation pictured below.

$$\begin{array}{lccccccc} (**) & t_1 & t_2 & \ldots & t_{j-2} & t_{j-1} & t_j & t_{j+1} \\ & ? & ? & \ldots & a & b & a & x \\ & & & & & & p_1 & p_2 \end{array}$$

We ask if $t_{j+1} = p_2$, i.e. "Is the $(j+1)^{st}$ text character equal to "b". If yes, then t_{j+2} is compared with p_3, otherwise the pattern is slid to the right once more, this time by one position. To understand the last observation, we note that our machine was in state one, indicating that "a", the first pattern character has successfully been matched with the text. But $x \neq$ "b", therefore, we traverse the failure link at position one. $F(1) = 0$, hence, a

slide of $i - f(i) = 1 - 0$ or 1 place to the right is indicated. Below, we give one last snapshot of this matching process.

$(***)$	t_1	t_2	...	t_{j-2}	t_{j-1}	t_j	t_{j+1}
	?	?	...	?	?	?	a
							p_1

We wish to determine if t_{j+1} equals "a". But we obtained the answer to this question several moments ago! $(in(*))$. Using the failure function to govern the length of pattern slides, is not giving our algorithm optimal behavior. We shall comment on this observation shortly.

We proceed with the computation of the failure function. The function f defined for $i = 1, 2, \ldots m$ is defined by :

$$f(i) = max\{k |\ p_1 p_2 \ldots p_k = p_{i-k+1} \ldots p_i\}$$

We let $f^0(i) = i$ and $f^{r+1}(i) = f(f^r(i))$, i.e. $f^r(i)$ denotes the composition of f with itself r times.

We set $f(1) = 0$ and

$$f(i+1) = \begin{cases} f^r(i) + 1 & \text{for } 0 < i < m, r \text{ is the smallest positive} \\ & \text{integer such that } p(f^r(i) + 1) = p_{i+1} \\ 0 & \text{if } f^r(i) = 0 \ \textit{and} \ p_{i+1} \neq p_1 \end{cases}$$

To gain some insight into this function, we focus our attention on the unfilled portion of table 1.2.

PATTERN:	a a b a a c a a b a a a
INDEX i:	1 2 3 4 5 6 7 8 9 10 11 12
f(i) :	0 1 0 1 * 0 1 2 3 4 5 *

Table 1.2: Failure function for the pattern *aabaacaabaaa*.

We need to compute $f(5)$. In this case, $i = 4$. We compare $p_{f^1(4)+1}$ with p_{4+1}. But $f^1(4) = 1$. So $p_{1+1} = p_2$ is being compared with p_5. Is "a"= "a"? Yes. Therefore, $f(5) = f(4) + 1 = 1 + 1 = 2$. That $f(5)$ equals 2 implies a suffix of the pattern of length two ending at position five, matches a prefix of the pattern of length two. That is $p_1p_2 = p_4p_5$, $aa = aa$.

We now wish to calculate $f(12)$. Here $i = 11$. So we compare $p(f^1(11) + 1)$ with $p(11 + 1)$. We have that $f(11) = 5$. $p(5 + 1) = p(6)$ must be compared with p_{12}, But "c"$\neq$ "a". The value of r will have to be incremented, which implies that the length of the sought correspondence (if indeed there is one) within the pattern is now shorter. We compare $p(f^2(11) + 1)$ with p_{12}. $f^2(11) = f(f(11)) = f(5) = 2$ (we just completed that calculation). We compare p_{2+1} with p_{12}. But $p_3 \neq p_{12}$, i.e., "b" $\neq$ "a". Incrementing the value of r one last time we obtain that $p_{f^3(11)+1}$ or p_2 is equal to p_{12}, i.e. "a" = "a". Therefore, $f(12)$ is set to 2. Of course, this implies that the aa at the beginning of the pattern matches the aa at its

right end.

We mentioned previously that using the failure function f to govern the length of pattern slides with respect to the text, may cause unnecessary comparisions to be made. Instead, we use the function next(i) which is similar to $f(i)$. If $P_{i+1} \neq P_{f(i)+1}$, then next$(i+1)$ equals $f(i+1)$. But, if $P_{i+1} = P_{f(i)+1}$ then next$(i+1)$ = next$(f(i+1))$. Next(1) would be defined as 0. To clarity this distinction between $f(i)$ and next(i) we consider the pattern *abaab* once more (see table 1.3). Suppose we have successfully matched *abaa* with the text, and the next text character is not a "*b*". It is foolish to follow the failure function which causes us to try to match the second pattern character (another "*b*") with the same text character. Using the next function instead, allows a slide of 4 positions to the right. For the details entailed here the reader is referred to [*KNUTH77*].

PATTERN:	a b a a b
INDEX i:	1 2 3 4 5
f(i) :	0 0 1 1 2
NEXT :	0 0 0 1 0

Table 1.3: Comparison of next with failure function for the pattern *abaab*.

A version of the KMP incorporating the next function appears below [*KNUTH77*]

$i \leftarrow 1; j \leftarrow 1;$
while $i \leq patlen$ **and** $j \leq textlen$ **do**
    begin
        while $i > 0$ **and** $text(j) \neq pattern(i)$
            do $i \leftarrow next(i)$;
        $i \leftarrow i + 1$; $j \leftarrow j + 1$
    end ;

The complexity of this algorithm is naturally proportional to the number of character comparisons required. This will always be linear in the sum of pattern length and text length. The reader is referred to [*KNUTH77*] for a discussion of changes to this algorithm to improve its efficiency. Modifying the KMP algorithm to find all occurrences of a pattern in a text is straightforward. The details are in the same reference. The reader wishing to see several more examples of the KMP flowchart construction are referred to an excellent introductory text by Sara Baase on Algorithmic Complexity [*BAASE79*]. A recent book by Sedgewick [*SED83*] also contains several excellent chapters on this material.

CHAPTER 2

BOYER MOORE ALGORITHM

§2.1 *Boyer Moore – a description and the algorithm*

In the KMP algorithm, string matching proceeded in an essentially left-to-right fashion. Boyer and Moore [*BOY77*] have provided an algorithm which searches for pattern in text by beginning at the right end of pattern. See below :

$$(1) \qquad \begin{array}{cccccccc} p_1 & p_2 & \cdots & p_{m-1} & p_m & & & \\ & & & & | & & & \\ t_1 & t_2 & \cdots & t_{m-1} & t_m & t_{m+1} & \cdots & t_n \end{array}$$

That is, we begin by comparing p_m with t_m. If these characters match, p_{m-1} is compared with t_{m-1}, and so on. If m consecutive successful matches occur, then there is a match, at the position of the m^{th} character comparison

(e.g., at t_1 in (1) above). If, however, a mismatch occurs, then we need to slide the pattern right with respect to the text. Two pre-computed tables are consulted, to determine the maximal permissible slide to the right.

Suppose in (1) above, we had obtained a mismatch on the first character comparison (i.e., it was discovered that $p_m \neq t_m$. We adopt the notation of reference [*BOY77*]. We denote by *char* that character in text which caused the mismatch. Essentially, two cases arise (actually, the first is just a special instance of the second). The first is that *char* does not occur anywhere in pattern. Here, we may slide the pattern patlen places to the right (the maximal possible). A slide of length less than patlen will ensure a mismatch, since *char* will be aligned with a character from pattern which it cannot possibly match. More generally, we ask : What is the rightmost occurrence of *char* in pattern ? Consider (2) below :

(2)

p_1	p_2	p_3	p_4	p_5	p_6	p_7
s	y	n	e	r	g	y
?	?	?	?	?	?	s
t_1	t_2	t_3	t_4	t_5	t_6	t_7
						↑

Question marks indicate text characters not yet inspected. The vertical arrow (below t_7) points to the current comparison being made. In this example we have that *char* (an "s") is not equal to p_7 which is a "y". The rightmost occurrence of *char* in pattern is at position one (and no other "s"'s occur in pattern to the right of p_1). After the slide, we want p_1 to

be above t_7, i.e., we want the two "s"' s to coincide. Therefore, we slide pattern to the right by patlen – the rightmost occurrence of *char* in pattern, or simply $7 - 1 = 6$ positions. The following configuration results (3).

(3)

							p_1	p_2	p_3	p_4	p_5	p_6	p_7
							s	y	n	e	r	g	y
							s	?	?	?	?	?	?
t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	
												↑	

Notice, that after the shift, our attention is once more focused on the rightmost character in pattern (p_7 is being compared with t_{13}). The amount of shifting, which is a function of characters from the alphabet, and their rightmost occurrence within pattern is called $delta_1$. $Delta_1$ is precomputed once at the beginning of the search; a table is constructed, one entry per alphabet symbol, which specifies appropriate shifts depending on *char*. Implementing $delta_1$ is linear in pattern length plus the size of the alphabet. For very large alphabets, the $delta_1$ table may be implemented as a hash array. This will of course impair the algorithm's asymptotic behavior.

We consider two more examples involving the use of $delta_1$. Suppose we have the situation depicted in (4) below.

(4)

p_1	p_2	p_3	p_4	p_5	p_6	p_7
s	y	n	e	r	g	y
				s	g	y
t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+5}	t_{i+6}
				↑		

p_7 and p_6 have been found to match t_{i+6} and t_{i+5} respectively. But *char* (i.e., t_{i+4} which is an "s") is seen not to equal p_5. We observed previously that the rightmost occurrence of "s" in pattern was at position one. We have already matched $l = 2$ characters successfully. We slide pattern to the right by $k = delta_1\ (char) - l$ positions. $Delta_1(s) = 7 - 1$ or 6. Therefore, our slide is of size $6 - l$ or $6 - 2$ which equals 4 positions to the right. At which point, an attempt would be made to match p_7 with t_{i+10}. That is, our attention would be focused at 6 or $delta_1$ positions to the right.

Finally, suppose we have the configuration shown below in (5).

(5)

p_1	p_2	p_3	p_4	p_5	p_6	p_7
s	y	n	e	r	g	y
				y	g	y
t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+5}	t_{i+6}
				↑		

We have that *char* equals "y". But, the rightmost occurrence of *char* in pattern is to the right of the current pattern position. $Delta_1(y) = 7-7 = 0$. This would dictate a slide of $k = delta_1(y) - l$ which equals $0 - 2$ or -2 positions to the right, which means 2 positions to the left. This prospect is unappealing ! $Delta_1$ is said to be useless in this situation, and instead we

slide the pattern one position to the right (always a safe strategy).

Boyer and Moore [*BOY77*] remark that the first version of their algorithm incorporated shifting using only $delta_1$ as described above. The average behavior of this approach (which we label as BM1) was quite similar (i.e., "sublinear") to their refined versions which are described below. However, the worst case behavior of BM1 is quadratic in the length of pattern and text. They offer, by way of example, the pattern $CA(BA)^r$ and any text of the form $((XX^rAA(BA)^r)^*$. This search will exhibit the quadratic time bound.

A later version of their algorithm (which we call BM3) incorporated a second strategy for shifting as well. Consider (6) below.

(6)

p_1	p_2	p_3	p_4	p_5	p_6	p_7
r	*o*	*c*	*o*	*c*	*c*	*o*
?	?	?	?	*o*	*c*	*o*
t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+5}	t_{i+6}
				↑		

p_6 and p_7 match t_{i+5} and t_{i+6} (i.e., $co = co$) That portion of pattern which coincides with corresponding text characters, we call subpat. In (6) above, *char*= "o" and p_5 = "*c*", a mismatch.We look for a reoccurrence of subpat (i.e. "co") to the left of the present one (by the least amount) which is not preceeded by the character in pattern which preceeds subpat. This is called a plausible reoccurrence. Subpat in our example ("co") is preceeded by a "c", the fifth character in pattern. Notice that p_3p_4 also equals "co".

This reoccurrence of subpat is preceeded by an "o" in position two.

Originally, Boyer and Moore did not require that the character preceeding the reoccurrence of subpat, be different than the character to the left of subpat. In this case, unnecessary comparisons would be made. To see this, refer to (7) below.

(7)

p_1	p_2 p_3	p_4	p_5 p_6
c	c o	c	c o
	reoccurence of subpat		*subpat*

Suppose p_4 causes a mismatch with some text character t_i . Sliding pattern to the right by three positions will bring another "c", the one in p_1, in alignment with t_i. Obviously, this is not optimal. Boyer and Moore attribute the subsequent improvement to Ben Kuipers of M.I.T. and Donald Knuth of Stanford. Observe the similarity here, with that of the definitions for the failure and next functions in the KMP algorithm.

This function based upon the difference between subpat and its rightmost plausible reoccurrence (rpr), is called $delta_2$. Actually $delta_2$ is equal to the difference between the position of subpat and its rpr (call this k), plus the length of subpat. We have $delta_2 = k + |subpat|$. To illustrate this definition, we consider once more the example in (6) which is reproduced

below.

p_1	p_2	p_3	p_4	p_5	p_6	p_7
r	o	c	o	c	c	o
				c	c	o
t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+5}	t_{i+6}
				↑		

Subpat equals "co". The length of subpat is two. The rpr of subpat occurs at p_3p_4. Hence $k = 6 - 3$ or 3. We slide pattern 3 positions to the right with respect to text. This is shown in (8) below.

(8)

			p_1	p_2	p_3	p_4	p_5	p_6	p_7
			r	o	c	o	c	c	o
				o	c	o	?	?	?
t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}	t_{i+5}	t_{i+6}	t_{i+7}	t_{i+8}	t_{i+9}
									↑

The rightmost character of pattern is being compared with t_{i+9}. Our attention has shifted $k + |subpat|$ or $3 + 2$ which equals 5 positions (the value of $delta_2$) to the right. To utilize the Boyer-Moore algorithm we slide the pattern the maximum of the three quantities: $1+|subpat|$, $delta_1$ (char) or $delta_2(j)$. It might be useful to consider an entire example ((9) below).

§2.2 A complete example

(9) PATTERN: VARY

TEXT: HURRY, WORRY, UNWARY VISITOR, NEVER VARY[1]

↑

A mismatch occurs immediately. *Char* which is an "R" occurs at position 3 in pat. $Delta_1$ dictates a slide of *patlen* − 3 or one position to the right.

PATTERN: VARY

TEXT: HURRY, WORRY, UNWARY VISITOR, NEVER VARY

↑ ·· ↑

We obtain two successful matches, until the leftmost "R" in HURRY fails to match the "A" in VERY. Notice that here $delta_1$ is useless (i.e. *char* occurs in pat to the *right* of the current comparison). We have successfully matched "RY", hence subpat has a length of two. But no rightmost reoccurrence of subpat occurs. If an occurrence could occur, it would be entirely to the left of VARY. Hence $delta_2$ permits a slide of 4 positions to the right, and our attention is shifted down string by 6 positions.

[1]from "Old Amusement Park", a poem by Marianne Moore in The Complete Poems of Marianne Moore, Maxmillan Co. 1967, p.210.

PATTERN: VAR Y

TEXT: HURRY, WORRY, UNWARY VISITOR, NEVER VARY

↑

"O" does not occur in pat. *Delta*$_1$ allows a slide of four places.

PATTERN: VARY

TEXT: HURRY, WORRY, UNWARY VISITOR, NEVER VARY

↑

Char is a comma which does not occur in the pattern. Once again *delta*$_1$ permits a slide of 4 places to right.

PATTERN: VARY

TEXT: HURRY, WORRY, UNWARY VISITOR, NEVER VARY

↑

"Y" does not match "W". *Delta*$_1$ applies here as well, we slide the pattern four positions.

PATTERN: VARY

TEXT: HURRY, WORRY, UNWARY VISITOR, NEVER VARY

↑

The blank does not match "Y" and does not occur in pattern, so once again $delta_1$ permits a slide of 4 positions to the right.

```
PATTERN:                                VARY
TEXT:     HURRY, WORRY, UNWARY    VI S ITOR, NEVER VARY
                                        ↑
```

Since "I" does not match "Y" and does not occur in pattern, we employ $delta_1$ for another maximum slide of size four.

```
PATTERN:                                VARY
TEXT:     HURRY, WORRY, UNWARY VISITOR, NEVER VARY
                                           ↑
```

$Delta_1$ dictates a slide of four positions because char, a comma, does not occur in pattern.

```
PATTERN:                                VARY
TEXT:     HURRY, WORRY, UNWARY VISITOR,  NEVER VARY
                                           ↑
```

The "V" in NEVER does not match the "Y" in VARY. However, there is a "V" in VARY, hence only a slide of 3 positions is permitted.

```
PATTERN:                                          VARY
TEXT:    HURRY, WORRY, UNWARY VISITOR, NEVER VARY
                                                  ↑
```

Blank does not occur in VARY, so we may slide 4 positions to the right.

```
PATTERN:                                          VARY
TEXT:    HURRY, WORRY, UNWARY VISITOR, NEVER VARY
                                                  ↑ · · ↑
```

A match is detected! The Boyer-Moore algorithm required 16 comparisons in this example to scan through 40 text characters. Less than one comparison was required for each character in text passed over. Boyer and Moore (in [*BOY77*]) empirically test their algorithm on random patterns, varying in length from one to fourteen. They verify the sublinear behavior observed above. The authors in [*BOY77*] observe that the number of comparisons (and the number of machine language instructions executed) required for each character passed, decreased as the pattern length increased. They

remark that for an English pattern of length 5, about .24 characters were inspected for every character passed, i.e. the average size of a slide to the right would be four.

§2.3 Several Improvements

Horspool [*HOR*79] proposes an algorithm based on the strategy of first searching in text for the character in pattern with the lowest probability of occurrence. The probabilities required for the correct operation of this approach may be established empirically (through random sampling) once, when the text is created. For example, suppose we are to search through an English text for the string "proximity". Horspool's algorithm would begin by searching through the text for an occurrence of "x" (which has a very low probability of occurrence in English). For patterns of length twelve, Horspool's algorithm attained a search rate of 5.6 million characters per second. This contrasts with a speed of 3.8 million characters per second for a more straightforward approach.

Horspool also tests two versions of the Boyer-Moore algorithm empirically. The first, which he calls BM, contains both the $delta_1$ and $delta_2$ functions. The second he labels SBM, for shorter Boyer-Moore, omits the $delta_2$ function. For patterns of length twelve both versions searched through a text at a rate of 9.4 million characters per second. Horspool remarks that the purpose of $delta_2$ is to optimize the handling of repetitive patterns and hence to avoid a worst case quadratic running time. It is clear from his experiments that such repetitiveness is not a problem in English (as opposed to binary) texts.

The major thrust in Horspool's work seems to be to obtain wider ac-

ceptance, by software practitioners, for the Boyer-Moore algorithm. His empirical evidence shows that for patterns of length six or greater, the Boyer-Moore algorithm (even the simpler version without $delta_2$) outperforms both search instructions built into the computer hardware, and his own scan for lowest frequency character approach.

Galil, in [*GALIL*79] modifies the Boyer-Moore algorithm so as to eliminate the difficult to compute $delta_2$ function, while at the same time preserving the algorithm's linear behavior in the worst case. His construction depends on the concept of periodicity. Given two strings w and u, where u is a prefix of w; u is said to be a period of w if w is a prefix of u^k for some $k \geq 1$. Consider the strings: $w = ababa$ and $u = ab$. We observe that for $k = 3$, u is a period of w (i.e. the string $w = ababa$ is a prefix of the string u^3 or $ababab$. This definition is equivalent to requiring that $w = uv$ where v is a prefix of w. In the above example we would have that $w = ababa$ while $u = abab$ and $v =$ "a". W is said to be periodic if the size of its period is not larger than $w/2$ (which naturally implies that $k \geq 2$). Hence, the string $w = ababa$ in the above example is periodic. Consider $w' = aba$, We would still have that $u = ab$ is a period, however w' is not periodic.

Galil presents the periodicity lemma. This states that if w has periods of sizes P and Q, then w has a period of size g.c.d.(P,Q), when $|w| > P+Q$. Several other lemmas are stated and proved. He then defines the concept of *closeness*. Two occurrences of the pattern at positions p and q are said

to be *close* whenever their distance from one another is less than or equal to half the pattern length. A neighborhood is taken to be the reflexive transitive closure of the relation *closeness*. Then a chunk is defined as a maximal set of neighbors. Galil goes on to prove that there is a lower bound on the distances between chunks (at least $m/2$), and on the number of such chunks (n/m). Galil's version of the Boyer-Moore algorithm seeks to discover chunks of occurrences of the pattern at one time, rather than single instances. When pat does not occur in text, Galil's approach is identical to the Boyer-Moore algorithm. When pat is found at location j, Galil's algorithm searches for the next occurrence at $j + k$ (where k is the length of the smallest period in pat). The worst case of this algorithm is shown to be linear in text length even when the pattern is periodic. When searching through English texts, the original Boyer-Moore algorithm is probably preferable. Galil's algorithm, however, should be given serious consideration if the text and pattern strings are binary in nature.

CHAPTER 3

GALIL'S ALGORITHM

§3.1 *Preliminaries*

Suppose that one has a parallel algorithm which uses p processors to solve some problem in time t. The cost c for such an algorithm is defined as the product of the number of processors employed and the amount of time required for a solution, that is, $c = pt$. Any parallel algorithm of time t and cost $c = pt$ may be converted to a serial algorithm (one employing a single processor) of time pt. The essential idea in the translation is that the lone processor simulates in time t the work of each of the p processors. A serial algorithm is deemed optimal if it can be shown that any other algorithm to solve the problem under consideration cannot possibly have a lower time complexity.

Whenever the *cost* of a parallel algorithm equals the *time complexity* of an optimal serial algorithm, the parallel algorithm is itself deemed to be optimal. Rivest has shown that any serial algorithm to solve the string

matching problem must require linear time in the worst case [*RIV*77]. Hence the Knuth-Morris-Pratt and Boyer-Moore algorithms described in chapters one and two are each optimal.

Zvi Galil [*GALIL*83*b*, 85] describes several parallel string matching algorithms. Two parallel versions of the Random Access Machine (RAM) are employed. One is a WRAM, a parallel computer with p RAMs, and a shared common memory. It is a synchronous machine in which simultaneous READs and WRITEs are permitted. In the case of a parallel WRITE only a simultaneous AND instruction is allowed. The other model of computation is a PRAM in which only parallel READs may occur. Galil's main algorithm has parallel linear running time, i.e., the cost which equals pt is $O(n)$. His was the first optimal parallel algorithm for string matching.

Galil's algorithms are not derived from either of the linear time algorithms discussed in chapters one and two. The KMP and Boyer-Moore algorithms do not appear to be parallelizable, for as Galil states :" They construct sequentially tables which are used sequentially. So, even giving the tables for free does not seem to help much."[2] His algorithms employ properties of periodicities of strings. Periodicity was discussed briefly in chapter two. However, to keep the present chapter self-contained we restate several definitions below and also present a few theorems without proof. Instead, examples which should appeal to one's intuition are pro-

[2]Galil,"Optimal Parallel Algorithms For String Matching" Technical Report, Columbia University, 1983.

vided. The reader familiar with periodicity in strings may safely skip the next few paragraphs (until section 3.2).

A string u is said to be a period of a string w if w is prefix of u^k for some k. This is equivalent to requiring that w be a prefix of uw. Consider the string $w = ababababa$. Each of *ab*, *abab* and *ababab* are periods of w. The shortest period of a string is called *the* period of w. Also, the period size is defined as the length of the period. The period for w is *ab*, its period size being $|ab|$ or two. A string w is said to be periodic if it is at least twice as long as its period. The string $w = ababababa$ from our example is periodic.

Galil's algorithms consider prefixes of the pattern x of increasing length. At each stage in the algorithm, the length of the prefix doubles (approximately). If a prefix is periodic, we are concerned if the periodicity continues into subsequent and hence larger prefixes. Consider the strings $u = abab$, $v = abababab$ and $w = ababababc$. Here we have that the periodicity of u continues in v but terminates in w. Galil presents the following six facts on periodicities, proofs of which may be found in the aforementioned references.

FACT 1: (The Periodicity Lemma) : If w has two periods of sizes P and Q, with $|w| \geq P + Q$, then the string w has a period of size $g.c.d.(P, Q)$. Let $w = abcbabcbabcbabcbc$ then w has periods of size 8 and 12. FACT 1 claims that a period of length $g.c.d.(8, 12)$ or 4 must exist. The string $u = abcb$ is such a period for w.

FACT 2: If v occurs at positons j and $j + P'$, where $P' \leq |v|/2$ then: 1) v is periodic and there is a period of length P', and 2) v occurs at $j + P$, where P is the period size of v. Let $z = cababababababa$ and $v = ababababa$ with $P' = 4$. The string v occurs at positions 2 and 6. One can readily verify that both assertions hold.

The remaining facts deal with a string v which is assumed to be periodic, i.e., $v = u^k u'$ with $k > 1$. u' is a proper prefix of u where u is the period of v with $|u| = P$. Let $L = lP$ where $l = \lceil |v|/P \rceil$.

FACT 3: If v occurs at j and $j + mP$ with $m \leq k$, then $u^{k+m}u'$ occurs at j. Let $z = ddacacacacacacaca$ and $v = acacaca$. Here, k has a value of 3 and P is equal to 2. We choose $m = 2$, v occurs in z at positions 3 and $3+2\times2$ or 7. Notice that $(ac)^{k+m}a$, i.e., $(ac)^5a$ occurs at position 3.

FACT 4: v occurs at $j, j + P$ and $j + L$ iff $u^{k+l}u'$ occurs at j. We employ the strings v and z from FACT 3. v occurs at position 3, $3 + P$ or at position 5 and at position $3 + L$ (where $L = 4 \times 2$ or 8) which is position 11. Observe that $u^{k+l}u'$ occurs at position 3.

FACT 5: If v occurs at positions j and $j + \Delta$ where $\Delta \leq L - P$, then Δ must be a multiple of P. Once more the strings v and z from FACT 3 are used. It is required that Δ be no larger than $L - P$ or 6. Let $\Delta = 4$, the string v occurs at position 3 and at position $3 + 4$ or 7.

The quantity Δ is equal to $2P$.

FACT 6: An occurrence of the string v at position j is said to be *important* if v does *not* occur at $j + P$. If there are two important occurrences of v at r and s, with $r > s$, then $r - s$ must be greater than $L - P$. If $r - s$ were less than or equal to $L - P$ then by FACTS 3 and 5 we would have that the occurrence at s is *not* important.

§3.2 *A description of the algorithm*

The input to Galil's algorithm is a string $Z = X\$Y$. X is the pattern of length n and Y is the text string. The length of the text in the preliminary version of the algorithm is $2n$. Hence $|Z| = 3n + 1$. Each of the strings X and Y are over a fixed size alphabet not containing \$.

The output is a Boolean array of length $3n + 1$ called $SWITCH$. The final output of $SWITCH(i) = 1$ iff an occurrence of X starts with Z_i (the i^{th} character in Z). The hardware requirement for this version of the algorithm is $3n + 1$ processors. Processor i is responsible for Z_i and $SWITCH(i)$.

Some terminology is in order. Given a string u of length l, one tests for u at position i of Z when the following operation is executed :

$$AND(u_1 = Z_i, \ldots, u_l = Z_{i+l-1}).$$

On a WRAM this test requires one unit of time. Galil remarks that a straightforward algorithm to test for X at all i's would require n^2 processors. Instead, one checks for prefixes of X in Z of increasing length. We let $X^{(i)}$ denote a prefix of X of size 2^i. And $X^{(i+1)}$ will represent a prefix of X of size 2^{i+1}, its general form will be $X^{(i)}Y^{(i)}$

There are $\log n$ stages in the algorithm. $SWITCH(j)$ will equal 1 after stage i iff $X^{(i)}$ occurs at j. Stage $i+1$ tests whether each occurrence of $X^{(i)}$ is followed by an occurrence of $Y^{(i)}$. If not, then a 1 in $SWITCH$ will be turned off. $SWITCH$ is divided into blocks of size 2^{i-1}. Therefore, in stage $i+1$, checking for $Y^{(i)}$ which is a string of length 2^i will require two time units. Property i is said to hold if each block has at most one 1. Before presenting a flowchart and working out several examples, it might be helpful to first give a quick overview of the algorithm. There are two cases to be considered. In the *regular case*, the first new block of $SWITCH$ has only one 1. A moment's reflection indicates that the 1 must occur at position one, (recall the format of Z). The other possibility is that the first new block may contain two 1's, the second of which occurring at position $P+1$. Galil refers to this as the *periodic case*. That is, $X^{(i)}$ is periodic with period size P (refer to FACT 2).

In the regular case in which the first block of $SWITCH$ contains a single 1, the other blocks may have at most two 1's. In a block with two 1's, the 1 at the smaller position is turned off. That's because this particular

occurrence of $X^{(i)}$ is *not* a beginning of an occurrence of $X^{(i+1)}$. When all blocks of $SWITCH$ have been scrutinized in a similar fashion then property i will hold. Then, as we commented earlier, the 2^{i-1} processors responsible for each block can check for $Y^{(i)}$ at all indicated positions in two time units.

In the periodic case when the first block has two 1's, we need to test whether the periodicity of $X^{(i)}$ continues in $X^{(i+1)}$. If $X^{(i+1)}$ has the same period, we will find all its occurrences and then enter stage $i+2$ in the periodic case. If $X^{(i+1)}$ does not have the same period, then many of the 1's in $SWITCH$ are turned off and the stage is completed with a regular step.

A bulletin board **BB** is employed by the algorithm for global communication. If a particular case is periodic then this fact is noted in the BB and the size of the period is also displayed. For local communication the algorithm uses local bulletin boards *lbbs*. The processors responsible for a block need to communicate to discover which comparisons should be made. Galil suggests that the last processor in each block may act as a *lbb*. At the end of each stage, one of every two *lbbs* dies. Before "passing away" a *lbb* transfers information to a neighboring *lbb*.

The flowchart for Galil's Algorithm appears in figure 3.1. Box 1 is a check to determine whether a particular stage is periodic or not. If yes, then boxes 5, 6 and 7 are responsible for checking if the periodicity continues into larger prefixes. The right portion of the flowchart handles the regular case.

When property i holds, ensuring that each block of $SWITCH$ has at most one 1, then a straightforward check for occurrences of $Y^{(i)}$ after each $X^{(i)}$ is carried out in the regular step (Box 4).

§3.3 *Details of the algorithm*

We consider several examples to help illustrate this algorithm. The first will clarify Boxes 2,3 and 4. We let the pattern $X = abaa$. The text Y is chosen as *abababaa*. Consequently, the string $Z = abaa\$abababaa$. To begin, each processor P_j where $1 \leq j \leq 3n + 1$ tests whether $Z_j Z_{j+1} = X_1 X_2$. The algorithm is checking for prefixes of length two in Z, i.e., for all occurrences of *ab*. Whenever a test succeeds at position j then P_j turns on $SWITCH(j)$ and makes the j^{th} *lbb* for the second stage point to the 1. After stage one, the array $SWITCH$ will appear as on page 41. Z is also presented for clarity.

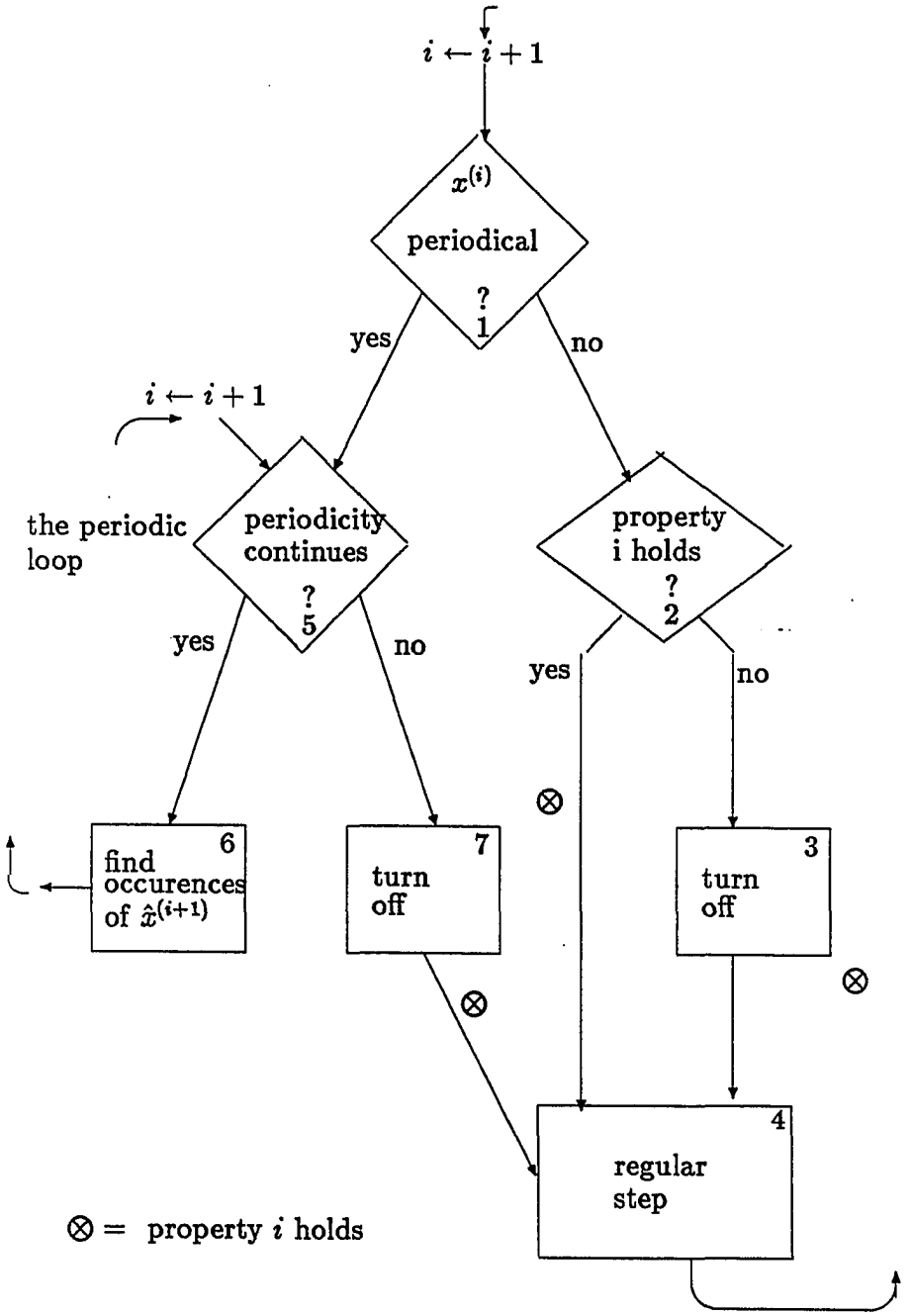


Figure 3.1: Flowchart for Galil's Algorithm, Stage $i+1$.

Z

1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	a	a	$	a	b	a	b	a	b	a	a

SWITCH

1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	0	1	0	1	0	1	0	0	0

lbb

1	2	3	4	5	6	7	8	9	10	11	12	13
0					5		7		9			

We observe that $SWITCH$ contains four 1's, at locations $1, 6, 8$ and 10. This indicates that $X^{(i)}$, the prefix of length 2 in X (which is just the string ab) may be found at the specified positions in Z. That is $Z_1Z_2 = X_1X_2 = ab$ (of course !), and each of Z_6Z_7, Z_8Z_9, and $Z_{10}Z_{11}$ also equals ab.

Recall that the size of a block when we begin stage $i+1$ is 2^{i-1}. So that as stage two commences, indicating $i = 1$, each block is of size one, i.e., contains a single processor. In illustrating the action of stage one, note that we have also included the contents of *lbbs* (only the contents of those containing useful information). As stage two begins the first two *lbbs* communicate (i.e., processors 1 and 2). An attempt is made to determine if $X^{(i)}$ is periodical (see Box 1). These two *lbbs* contain only a single 1 between them, hence stage 2 is a regular case. Leaving Box 1 in the flowchart, we enter Box 2 where a check for property i is made. At this point, the size of a block is one, hence property i holds in a vacuous sense. Therefore, we

may proceed directly to Box 4.

The regular step specified in Box 4 checks for prefixes of the pattern of length four that occur in the Z array. The algorithm will only concern itself with positions in Z where prefixes of length two are known to exist, i.e., those locations i where $SWITCH(i) = 1$. Consequently, only processors $1, 6, 8$ and 10 will be activated. The contents of a *lbb* is denoted by Δ. When there is a one present at position j of $SWITCH$, we let the *lbb* for the block containing processor j point to $j - 1$. The *lbbs* for blocks $1, 6, 8$ and 10 will therefore contain the integers $0, 5, 7$ and 9 respectively (refer to the prior display of $SWITCH$).

The four processors involved in this step will each make two comparisons. Processor j in the block will check if :

$$X_k = Z_{k+\Delta} \text{ for } K = \{j + 2^i, j + 2^i + 2^{i-1}\}$$

where $i = 1$. Note that j refers to the number of a processor within a block. Since blocks are currently of size one, each processor may of course be referred to as the first processor within its block. Therefore, for each processor indicated K will equal $\{1 + 2^1, 1 + 2^1 + 2^0\}$, i.e., $K = \{3, 4\}$.

In two time units, each of the processors in parallel will make the comparisions specified below.

Processor 1 checks if :

$$X_3 = Z_{3+0} \qquad // \text{ 0 is the contents of the first } lbb //$$

$$X_3 = Z_3$$

$$a = a \qquad yes$$

then:

$$X_4 = Z_{4+0}$$

$$X_4 = Z_4$$

$$a = a \qquad yes$$

Therefore $SWITCH(1)$ **remains set.**

Processor 6 makes the following comparisons :

$$X_3 = Z_{3+5}$$

$$X_3 = Z_8$$

$$a = a \qquad yes$$

then:

$$X_4 = Z_{4+5}$$

$$X_4 = Z_9$$

$$a = b \qquad no.$$

therefore, the 1 at $SWITCH(6)$ is turned off.

Processor 8 compares the a in the third position of X with the a in Z_{10}. However, in the next time unit it discovers that the 4^{th} symbol of pattern, an a, does not match the b in Z_{11}. Processor 8 then turns off $SWITCH(8)$. Meanwhile, processor 10 verifies that X_3 matches Z_{12} and X_4 matches Z_{13}. The output of the algorithm appears below. Naturally, a 1 will always occur at position one in $SWITCH$. The algorithm has determined that the pattern $abaa$ occurs once within text beginning at position 10 in Z.

SWITCH

1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	0	0	0	0	0	1	0	0	0

Next, we illustrate an example to shed light on the manner in which the algorithm handles periodicity. We let the pattern $X = a^5ba^3$ and the text Y is chosen as $a^5ba^5ba^5b$. Once again, the algorithm begins by attempting to discover prefixes of length two in the Z array. That is, processor P_j tests whether $Z_jZ_{j+1} = X_1X_2$. If the test succeeds, P_j turns on $SWITCH(j)$ and makes the j^{th} lbb for the second stage point to the 1. The array $SWITCH$ as it appears after stage one and the input Z are presented in figure 3.2. The contents of those lbbs which contain important information is included as well. **BB** is shown at the top of the next page, its contents will be explained in the next paragraph.

BB

P	L
0	0

We enter Box 1 in stage 2. The algorithm must determine if $X^{(i)}$, i.e. $X^{(1)}$ which is the string aa is periodical. The 2^{nd} processor looks at its *lbb* and notices that it is nonempty. This processor knows that the *lbb* of processor 1 contains a 1, hence $X^{(1)}$ is periodical. Processor 2 posts the period size P which equals 1 on **BB**. The length of the prefix which is periodic L, is also posted. L equals lP with $l = \lceil 2^i/P \rceil$ or 2 in this case. The **BB** appears as below:

BB

P	L
1	2

Galil comments that l will equal 2 or 3 when entering Box 5 from Box 1. Let $\hat{X}^{(i+1)}$ be the prefix of X of size $2^i + L$. We have : $|X^{(i+1)}| = 2^{i+1} \leq |\hat{X}^{(i+1)}| < 2^{i+1} + P$. Here in stage two, $|\hat{X}^{(i+1)}|$ equals $2^1 + L$ which is 4, i.e., $\hat{X}^{(i+1)} = aaaa$.

In Box 5 we test whether the periodicity continues in $\hat{X}^{(i+1)}$ by using $X^{(i)}$ as a yardstick. We glance back at FACT 4 at this time. Letting $v = X^{(i)}$ and with $j = 1$, $\hat{X}^{(i+1)}$ is seen to equal $u^{k+l}u'$. Processor 1 tests whether $SWITCH(P+1) = 1$ and $SWITCH(L+1) = 1$. But each of $SWITCH(2)$ and $SWITCH(3)$ are equal to one, hence the periodicity

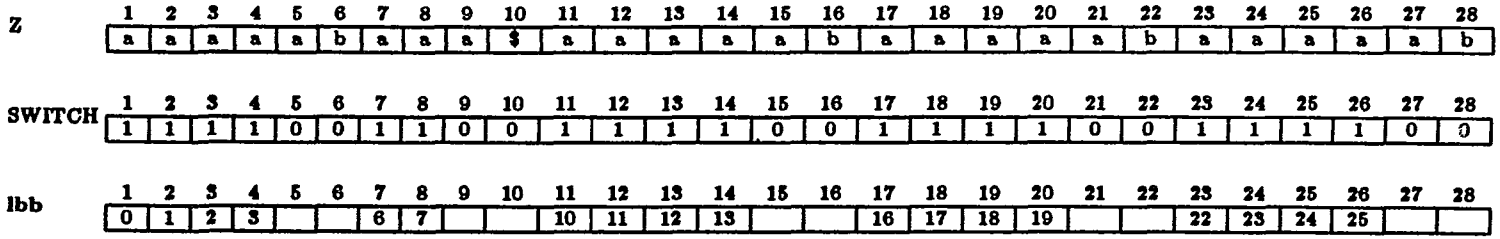


Figure 3.2: Contents of SWITCH and *lbb* registers after stage one is complete.

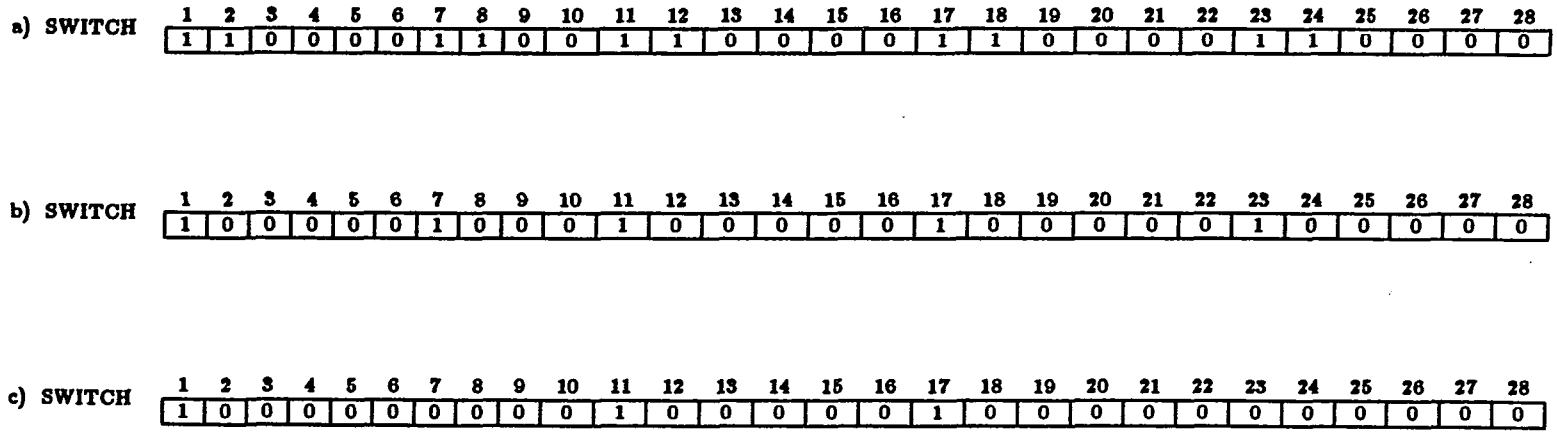


Figure 3.3: Subsequent updates of SWITCH array for our second example.

does continue. Observe that in this instance, the first test was redundant.

The algorithm proceeds to Box 6 where a search for occurrences of $\hat{X}^{(i+1)}$ is undertaken. Again, use is made of FACT 4 with $v = X^{(i)}$ and $\hat{X}^{(i+1)} = u^{k+l}u'$. If processor P_j sees a 1 at $SWITCH(j)$, then it checks whether $SWITCH(P+j) = 1$ and $SWITCH(L+j) = 1$. If one of the tests fails P_j turns off the 1.

Obviously, $SWITCH(1)$ remains on. Processor 2 checks $SWITCH(1+2)$ and $SWITCH(2+2)$. Since both $SWITCH(3)$ and $SWITCH(4)$ equal 1, $SWITCH(2)$ stays set. Processors 3, 4, 7, 8, 11, 12, 13, 14, 17, 18, 19, 20, 23, 24, 25 and 26 will check for occurrences of $\hat{X}^{(i+1)}$ or $aaaa$ in a similar manner. The reader may wish to verify a few cases for himself. The updated value of $SWITCH$ appears in figure 3.3(a).

The careful reader will have by now discovered an apparent inconsistency. The 1's at positions 7 and 8 in $SWITCH$ have *not* been turned off. And yet it is obvious that the prefix $aaaa$ of X cannot possibly occur at these locations due to the \$ in Z_{10}. If we have an occurrence of $X^{(i)}$ at position $j \leq n$ in stage $i+1$ and $j + 2^{i+1} > n+1$, then $X^{(i+1)}$ cannot possibly occur at j simply because this prefix is too long. Galil proscribes that if the first mismatch from the left is the \$, the 1 at $SWITCH(j)$ is not to be turned off. Hence a 1 in $SWITCH$ may denote an *overhanging occurrence* of $X^{(i+1)}$. If in the last stage either property i holds or the periodicity terminates, the algorithm terminates with a regular step and these 1's will

be removed. Otherwise the last step will involve computations similar to what occurs for important occurrences (to be described shortly).

Returning to the calculations, the algorithm is in the periodic loop and hence there is a return to Box 5. The value of L needs to be updated. If $2L - P > 2^{i+1}$ then $L \leftarrow 2L - P$, otherwise L is set to $2L$. In our example we have that $2L - P = 4 - 1 = 3$, this is $< 2^{1+1}$ or 4. Therefore, L is set to $2L$ and now has a value of 4. **BB** as it appears after this update is shown below.

BB

P	L
1	4

We are now beginning stage 3. The algorithm needs to determine if the discovered periodicity will continue into $X^{(3)}$, a prefix of pattern of length 2^3 or 8. FACT 4 dictates that processor 1 should check each of $SWITCH(1+P)$ and $SWITCH(1+L)$. Before proceeding, we reiterate a definition provided earlier in FACT 6. An occurrence of the string v at positon j is said to be *important* if v does not occur at $j + P$. When processor 1 checks $SWITCH(1+P)$ or $SWITCH(2)$, it is found to equal 1, hence the occurrence of $X^{(i)}$ at position one in Z, is *not* important. Had this occurrence at Z_1 been important, P_1 would have posted a 0 on **BB**. However, when P_1 checks $SWITCH(1+L)$ or $SWITCH(5)$, a 0 is

discovered and hence the periodicity does not continue. Furthermore, by FACT 6, it is known that there is a unique important occurrence of $X^{(i)}$ within the first $L+1-P$ or 4 positions of Z. At this point, each processor P_j (in this case $j = 1,\ldots,4$) where $SWITCH(j) = 1$ will search for this important occurrence. Processor P_2 discovers that the occurrence of $X^{(i)}$ at Z_2 is important, since SWITCH(2+P), or $SWITCH(3)$ equals 0. P_2 posts 1 on BB (in general P_j posts $j-1$).

Next, each processor P_k with $SWITCH(k) = 1$ uses the posted value of $j-1$ and the $SWITCH$ array to check for an important occurrence of $X^{(i)}$ at $k+j-1$. If the occurrence at position $k+j-1$ is found not to be important, then naturally the processor is justified in turning off the 1 at $SWITCH(k)$.

Processor P_2 checks if the occurrence at Z_3 is important. There is no occurrence of $X^{(i)}$ at Z_3 and so $SWITCH(2)$ is turned off. Processor P_7 verfies that the occurrence of $X^{(i)}$ at position 8 *is* important (since $SWITCH(8+P) = SWITCH(9)$ equals 0). Therefore $SWITCH(7)$ remains set. SWITCH(8), however, is turned off as it is not followed by an occurrence of $X^{(i)}$. Each of processors 11, 12, 17, 18, 23, 24 will perform a similar check. The occurrences of $X^{(i)}$ at positions 12, 18 and 24 are important and therefore the 1's in positions 11, 17 and 23 will remain set. To finish stage 3 of the algorithm, a regular step will be executed. As we enter Box 4 of the flowchart, $SWITCH$ will appear as shown in figure 3.3

(b).

Block size still equals two. And a quick check of $SWITCH$ indicates that Property i holds. But before executing the regular step, the local bulletin boards will have to be restored. Galil indicates that locations for *lbb*s can be maintained dynamically while we're in the periodic loop. In stage 3, *lbb*s will occur at locations $j + 2^{i-1}$, here $i - 1$ equals one and $j = 0, 2, 4, \ldots, 26$.

Processors 1,7,11,17 and 23 will be activated. They will locate instances of $X^{(i+1)} = X^{(i)}Y^{(i)}$ by determining if each occurrence of $X^{(i)} = aaaa$ is followed by an occurrence of $Y^{(i)} = abaa$. After this regular step is performed only the 1's in locations 1, 11, and 17 will remain set.

We return to Box 1. This is stage 4, the final stage of the algorithm. Block size now equals 4. It must be determined whether $X^{(3)}$, the prefix of pattern of length 8 is periodic or not. The first block contains a 1, at $SWITCH(1)$. However block 2 which contains processors 5 through 8 has no 1's. Hence, we proceed to step 2 where it is verified that property i does hold. Therefore, we proceed to Box 4 where a regular step is executed. The final value of $SWITCH$ is given in figure 3.3(c) The pattern occurs at locations 11 and 17 in the Z array.

§3.4 *Complexity results and some generalizations*

At the beginning of this chapter we defined the cost for a parallel algorithm to be the product of the amount of processors and time required, i.e., $c = pt$. The above algorithm employs $3n + 1$ processors and requires log time, exhibiting a cost of $O(nlogn)$ which is clearly *not* optimal. Galil shows how the hardware requirements may be reduced to $n/logn$ processors. He employs the Four Russians Trick [*AHO*74] to pack $logn$ symbols into a single number. Each processor is responsible for S consecutive symbols in the Z and $SWITCH$ arrays where $S = c \log n$ and c is a function of the alphabet size. This version of his algorithm has a cost which is $O(n)$ and is hence optimal.

Galil also discusses the implementation of this algorithm on a PRAM. Recall that on this parallel model, concurrent WRITEs are *not* permitted. Hence when 2^{i-1} processors of a block compute an AND as is the case during the regular step of the algorithm, a logarithmic amount of time (logarithmic in block length) will be required. Therefore on a PRAM, Galil's algorithm will require $O(log^2 n)$ time. However, by employing only $n/log^2 n$ processors and requiring that each processor be responsible for $log^2 n$ symbols, it is still possible to obtain an optimal cost algorithm.

Throughout this chapter, we have assumed that the text string was twice the pattern length. Galil also considers the case wherein $|X|$ and $|Y|$ are unrelated. Let m be the pattern length, i.e., $|X| = m$. And let n be

the length of the input, i.e., $n = |X| + |Y|$. To handle this situation we need merely divide the text into overlapping segments and let the processors search for occurrences within these segments. When the number of processors is small, i.e., $p \leq 2n/m$, one processor is assigned per segment and the problem is solved sequentially, certainly an optimal strategy. When the number of processors is large, i.e., $p > 2n/m$ but $p \leq n/\log m$, Galil also shows how to partition the text and by assigning several processors per segment still obtain an optimal algorithm.

CHAPTER 4

VISHKIN'S ALGORITHM

§4.1 *Description of the Algorithm*

Vishkin [$VISH85$] describes a parallel string matching algorithm which runs in O(n/p) time and employs $p \leq n/\log m$ processors, so if $p = n/\log m$ processors are engaged, Vishkin's algorithm exhibits $O(\log m)$ behavior ! (recall that m = pattern length) . His algorithm employs concurrent READ conconcurrent WRITE (CRCW) PRAMs, but Vishkin also describes an implementation on a CREW (concurrent READ exclusive WRITE) PRAM which runs in $O(\log^2 n)$ time.

The reader may recall that Galil's algorithm runs in $O(\log n)$ time and uses $n/\log n$ processors. It was required that the alphabet be of fixed size. And to obtain the optimal cost version of his algorithm, Galil employs the Four Russians Trick. Vishkin's algorithm does not require the latter ploy and furthermore places no restrictions on alphabet size. His work does, however, make generous use of the concepts of periodicity in strings. Indeed

for that matter, even the optimal sequential strategies, i.e., Boyer-Moore and Knuth-Morris-Pratt obtain their great speed to a large extent by the judicious handling of repetitive substrings.

A theorem of Brent's is useful in relating the amount of work a parallel algorithm must perform and the time it will require. His theorem states that any synchronous parallel algorithm of time t that consists of a total of x elementary operations can be implemented by p processors within a time of $\lceil x/p \rceil + t$. It becomes evident that the performance of Vishkin's algorithm will hinge upon an efficient assignment of processors to their jobs.

The algorithm consists of three steps. An analysis of the pattern is performed in the first step. The output of step one is used next to find a sparse set of "suspicious" indices of the text. A suspicious index in the text is one where an occurrence of the pattern may start. And finally in step three a character by character check is performed to determine which suspicious indices actually correspond to occurences of the pattern.

Suppose that $PAT[j, j+1, \ldots, m]$, where PAT is our shorthand for pattern, is not a prefix of $PAT[1, \ldots, m]$ for some j, $2 \leq j \leq m$. In this case an integer w will exist, $1 \leq w \leq m - j + 1$, such that $PAT(w) \neq PAT((j-1) + w)$. Vishkin refers to w as a witness to the mismatch and comments that w is a witness against the existence of a period of size $j - 1$ in the pattern. The existence of a period of size $j - 1$ in pattern requires that $PAT(w) = PAT(j - 1 + w)$. The definition of a period of size $j - 1$

implies that $PAT(i) = PAT(k)$ for all $1 \leq i \leq j-1$ with $k = (j-1)+i$. The output of step 1 will be an array of such witnesses, called WIT. That is, for each j where $2 \leq j \leq \lfloor m/2 \rfloor + 1$, step 1 determines whether the pattern has a period of size $j-1$, in which case $WIT(j) = 0$, and if not a witness is computed and assigned to $WIT(j)$.

It might be helpful if we consider a sample pattern and the appearence of its corresponding witness array. For the moment we will not concern ourselves with the details in step 1 involved in calculating this array.

	1	2	3	4	5	6	7	8
PATTERN	a	b	a	b	a	a	a	b
WIT	0	1	4	1	2	-	-	-

Observe that $WIT(1) = 0$. It should be obvious that for any pattern this will hold. $WIT(2) = 1$, this indicates that $PAT(2) \neq PAT(1)$ i.e., $b \neq a$. A glance at the WIT array indicates that $WIT(5) \neq 0$, hence we know immediately that the suffix commencing at position 5, i.e., *aaab*, is *not* a prefix of the pattern. The value of $WIT(5)$ is 2, therefore $PAT(2) \neq PAT(5-1+2)$, i.e., $b \neq a$.

The output of Vishkin's algorithm will be an array MATCH where $MATCH(i) = T$ when an occurrence of the pattern commences at position i in the text. For steps 2 and 3, we require the notion of k-blocks, where, for $k \geq 0$, the set of k-blocks is $\{TEXT[1,\ldots,2^k],\ldots,TEXT[l2^k + 1,\ldots,(l+1)\cdot 2^k]\ldots\}$. The processing of the text in each of these steps

depends upon whether or not the pattern is periodic. We will first consider the (easier) case in which the pattern is not periodic.

The initial phase of step 2 wherein the text is processed, will set (in parallel) $MATCH(i) = T$ for all i, $1 \leq i \leq n - m + 1$.

We need to define the *k-sparsity property* at this point; for each k-block there is at most one T value in $MATCH$. That is, each of $MATCH\{1,\ldots,2^k],\ldots,MATCH[l2^k + 1,\ldots,(l+1)2^k],\ldots\}$ contains a maximum of one T. The reader should note the similarity between Vishkin's k-sparsity property and Galil's property i. In step 2 we attempt to satisfy $(\lfloor \log m \rfloor - 1)-$ sparsity. After step 2 is completed, a T at $MATCH(i)$ will indicate that an occurrence of the pattern at position i of text is *possible*. It is the responsibility of step 3 to ascertain which of the T's in $MATCH$ actually correspond to occurrences of the pattern.

Steps 2 and 3 require a series of arrays, referred to as left arrays. $LEFT(k,a)$ will contain the entry of the leftmost T in $TEXT$ of k-block number a, where $1 \leq a \leq \lceil (n - m + 1)/2^k \rceil$, or an indication that no such T exists. Vishkin describes stage k of step 2:

stage k, $1 \leq k \leq \lfloor \log m \rfloor - 1$: Satisfy k-sparsity.

He notes that the input to stage k will satisfy $(k-1)-$ sparsity. Before proceeding, let's consider a sample text string and the corresponding $MATCH$ array as it would appear after initialization. Note that $MATCH(10)$

through $MATCH(16)$ are each set to F (which we have indicated with dashes). It is impossible for a pattern of length 8 to begin anywhere to the right of the 9^{th} position in a text of length 16.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TEXT	a	b	a	a	b	b	a	b	a	b	a	a	a	b	b	a
MATCH	T	T	T	T	T	T	T	T	T	-	-	-	-	-	-	-

The procedure below is performed in parallel for all k-blocks. Observe that as we begin stage 1, $k-1$ or 0-sparsity is satisfied (in a trivial manner because $k = 0$ implies a block size of one). We let a be an integer between 1 and $\lceil (n-m+1)/2^k \rceil$. Hence for our example where *text length* $= 16$ and *pattern length* $= 8$, a will assume values from 1 to $\lceil (16-8+1)/2^i \rceil$ or 1 to 5. The procedure is described for k-block a , where k-block a is the union of two $(k-1)$-blocks : $2a$ and $2a-1$.

if $LEFT(k-1, 2a) =$"null"
then $LEFT(k, a) \leftarrow LEFT(k-1, 2a-1)$
else **if** $LEFT(k-1, 2a-1) =$"null"
then $LEFT(k, a) \leftarrow LEFT(k-1, 2a)$
else ... see below ... ;

After stage $k-1$, $(k-1)$-sparsity will hold. Hence there is at most one index j_1 in $(k-1)-$block $2a$ and at least one index j_2 in $(k-1)-$block $2a-1$ such that $MATCH(j_1) = MATCH(j_2) = T$. The case which still needs to be explained is where both indices j_1 and j_2 exist. Vishkin employs the concept of a *duel* to eliminate one of these T's. Information contained in the array WIT will be employed toward this end. Refer to figure 4.1 below.

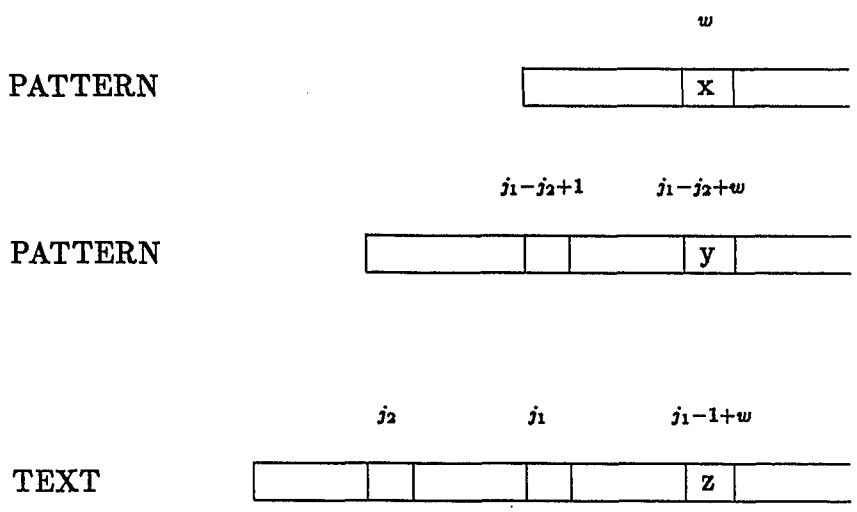


Figure 4.1: A duel between positions j_1 and j_2 of the text.

We let $j_1 > j_2$ be the two locations of text such that $j_1 - j_2 < m$ and the suffix starting at location $j_1 - j_2 + 1$ of the pattern is *not* a prefix of the pattern. Following step 1, position $j_1 - j_2 + 1$ of the witness table will point

to w, which is a counterexample. That is, $PAT(j_1 - j_2 + w)$ does not equal $PAT(w)$. Hence it is impossible for occurrences of the pattern to commence at both positions j_1 and j_2 of the text in each stage of step 2. Vishkin employs this duel concept to cut the number of T's in $MATCH$ in half. To be more precise, let w be $WIT(j_1 - j_2 + 1)$ and $z = TEXT(j_1 - 1 + w)$. We know that $j_1 - j_2 < 2^k$ because j_1 and j_2 are in the same k-block. When $k < \lfloor \log m \rfloor - 1, w \neq 0$ indicating that $PAT(j_1 - j_2 + 1, \ldots, m)$ is not a prefix of the pattern, that is $x \neq y$. If an occurrence of the pattern starts at j_1 then we have that $x = z$, however, if an occurrence of the pattern begins at j_2 then y must equal z. Since we know that $x \neq y$, only one of these two equalities can hold and hence at most one of the T's in $MATCH$ will survive. Vishkin's code for this duel appears below.

if $z \neq y$
then $MATCH(j_2) \leftarrow F$;

if $z \neq x$
then $MATCH(j_1) \leftarrow F$;

Finally,

if $MATCH(j_2) = T$

then $LEFT(k,a) \leftarrow j_2$

else if $MATCH(j_1) = T$

then $LEFT(k,a) \leftarrow j_1$

else $LEFT(k,a) \leftarrow$ "*null*";

For ease in the presentation, the sample pattern and witness array are redrawn below along with our text string and $MATCH$ array.

	1	2	3	4	5	6	7	8
PATTERN	a	b	a	b	a	a	a	b
WIT	0	1	4	1	2	-	-	-

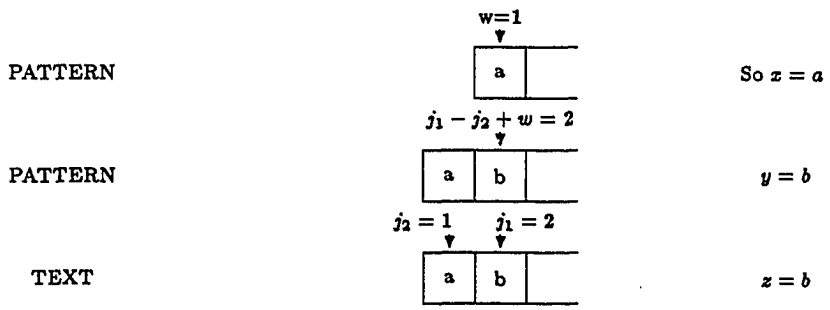
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TEXT	a	b	a	a	b	b	a	b	a	b	a	a	a	b	b	a
MATCH	T	T	T	T	T	T	T	T	T	-	-	-	-	-	-	-

§ *4.2 Processing of text for our example*

Observe that as we begin stage 1 of step 2, $LEFT(0,a) = T$ for $1 \leq a \leq 9$. Hence we will require 4 duels in parallel to decrease the number of T's to at most 5. We show the details for k-block 1, k of course being equal to 1 throughout this stage. Referring to figure 4.1 we have $j_2 = 1$ and $j_1 = 2$. W equals $WIT(j_1 - j_2 + 1)$ or $WIT(2 - 1 + 1)$, which is $WIT(2)$. In the

above table it is seen that $WIT(2) = 1$.

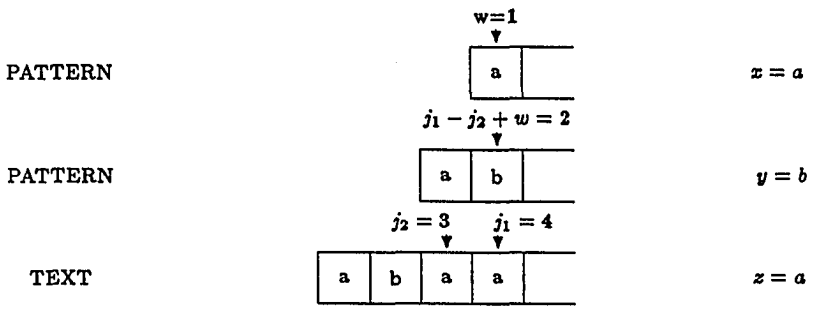
$X = PAT(w)$ or $PAT(1)$, hence $x = a, y = PAT(j_1 - j_2 + w)$ or $PAT(2 - 1 + 1)$, therefore $y = PAT(2)$ which is b. Finally, $z = TEXT(j_1 - 1 + w)$ or $TEXT(2 - 1 + 1)$, which is $TEXT(2)$ being a "b". Our duel is depicted below:



We have $z = y$, therefore, we cannot reset the T at $MATCH(j_2)$ to F. However, z not being equal to x permits us to eliminate the T at position j_1. Since $MATCH(j_2) = T$, $LEFT(1,1) \leftarrow j_2$ which is 1.

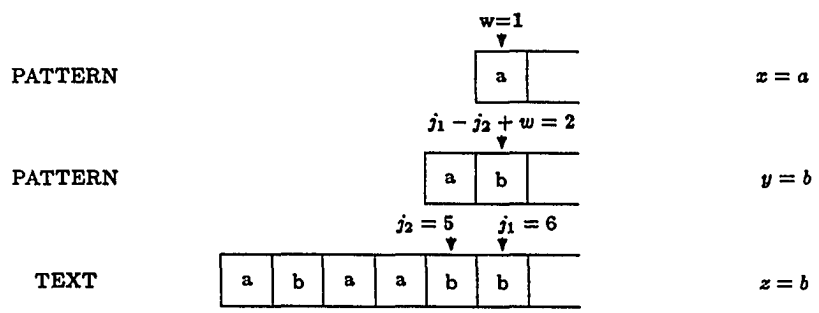
Consider k-block 2. $LEFT(0,3)$ and $LEFT(0,4)$ each contain a T, necessitating a duel here as well. Here, $j_2 = 3$ and $j_1 = 4$. W equals $WIT(4 - 3 + 1)$ which is $WIT(2)$. Referring to our witness array, w is seen to equal 1. X will equal $PAT(w)$ or $PAT(1)$ which is a. Y equals $PAT(j_1 - j_2 + w)$ or $PAT(2)$ which is b. Here, $z = TEXT(j_1 - 1 + w)$ or

$TEXT(4)$. Refer to the situation below.

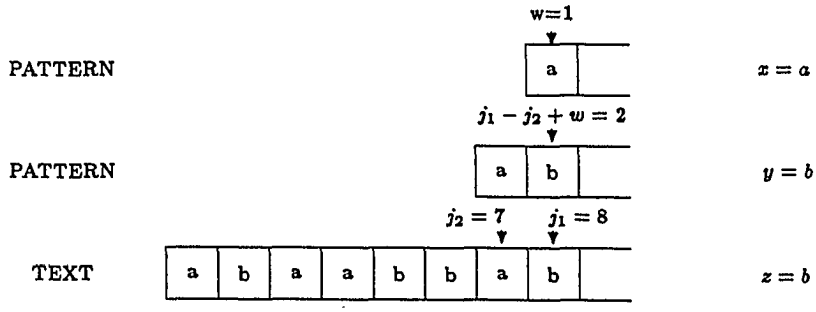


In the situation $z \neq y$ and therefore $MATCH(j_2)$ is reset to false and $LEFT(1,2) \leftarrow j_1$ or 4.

The 3^{rd} k-block produces the situation depicted below.



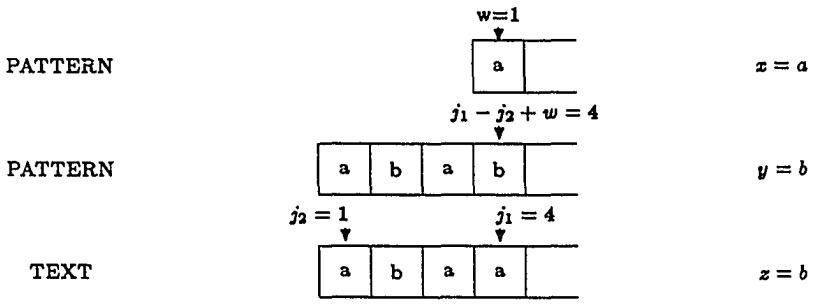
$MATCH(j_1)$ is set to F, and $LEFT(1,3) \leftarrow 5$. One last duel in this stage is required, in k-block 4 (see below).



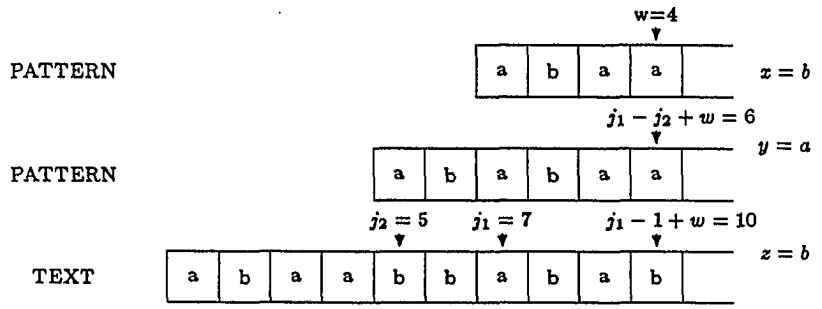
This case is similar to our prior duel. Hence $MATCH(j_1)$ is set to F and $LEFT(1,4)$ to 7. In k-block 5, we have that $LEFT(0,10)$ ="null" and therefore $LEFT(1,5)$ is set to $LEFT(0,9)$ which is 9. Our $TEXT$ is shown below along with the contents of the $MATCH$ and $LEFT$ arrays as they appear after stage one is complete.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TEXT	a	b	a	a	b	b	a	b	a	b	a	a	a	b	b	a
MATCH	T	F	F	T	T	F	T	F	T	-	-	-	-	-	-	-
LEFT(1,a)	1		4		5		7		9							
	(1,1)		(1,2)		(1,3)		(1,4)		(1,5)							

As we begin stage 2 of the second step, we observe that 1-sparsity is satisfied. That is, each 2-block of $TEXT$ contains at most one T. Here, a varies from 1 to $\lceil(16-8+1)/2^2\rceil$ or from 1 to 3. Referring once more to Vishkin's pseudocode for updating the $LEFT$ array, we notice that additional duels will be necessary. In k-block 1 (where $k=2$) we observe that $LEFT(1,1)=1$ and $LEFT(1,2)=4$, hence we set j_2 to 1 and j_1 to 4. The ensuing duel is shown below.



Clearly j_2 is the loser. $MATCH(j_2) \leftarrow F$ and $LEFT(2,1) \leftarrow j_1$ or 4. For k-block 2 we have $LEFT(1,3)=5$ and $LEFT(1,4)=7$, these become the values for j_2 and j_1 respectively. To determine the winner in this duel we consult the following sketch.



It is evident that an occurrence of pattern cannot begin at $TEXT(5)$ since $PAT(6) \neq TEXT(10)$ i.e., $y \neq z$. Therefore j_2 is set to F and $LEFT(2,2)$ to j_1 or 7.

Finally, for k-block 3 we have that $LEFT(1,6)$ = "*null*" and hence $LEFT(2,3)$ is set to $LEFT(1,5)$ which is 9. The updated $MATCH$ and $LEFT$ array is shown below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TEXT	a	b	a	a	b	b	a	b	a	b	a	a	a	b	b	a
MATCH	F	F	F	T	F	F	T	F	T	-	-	-	-	-	-	-
LEFT(2,a)	4				7				9							
	(2,1)				(2,2)				(2,3)							

Step 2 is complete once $(\lfloor \log m \rfloor - 1)-$ sparsity is satisfied. In our example the pattern length m is equal to 8, hence $\lfloor \log 8 \rfloor - 1$ or 2-sparsity must be satisfied. Referring to the above $MATCH$ array verifies that stage 2 is complete.

There are three T's in the $MATCH$ array. Step 3 will perform a character by character comparison to determine which of these T's should remain set. Vishkin's pseudocode for step 3 appears below.

Step 3. For each $\alpha, 1 \leq \alpha \leq n - m + 1$, such that $MATCH(\alpha) = T$ check character by character, if an occurrence of the pattern starts at α :

for all j, $1 \leq j \leq \lceil (n - m + 1)/2^{\lfloor \log m \rfloor - 1} \rceil$, **pardo**
  for all i, $1 \leq i \leq m$, **pardo**
    (Denote $t(j) = LEFT(\lfloor \log m \rfloor - 1, j)$)
    if $t(j) \neq$ "null"
    then if $TEXT(t(j) + i - 1) \neq PAT(i)$
      then $MATCH(t(j)) \leftarrow F$

Vishkin remarks that simultaneous WRITEs are possible in the last step. After step 3 is complete, $MATCH(i)$ will equal T if and only if an occurrence of the pattern commences at position i of the text. The text and

$MATCH$ array are shown as they would appear after step 3 is complete. Note that only one occurrence of the pattern occurs within the text beginning at position 7 as indicated by the T value of $MATCH(7)$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
TEXT	a	b	a	a	b	b	a	b	a	b	a	a	a	b	b	a
MATCH	F	F	F	F	F	F	T	F	F	-	-	-	-	-	-	-

§4.3 Periodic text considered

We remarked earlier that the form of the text analysis phase of Vishkin's algorithm depends upon whether the pattern is periodic or not. In our prior example the pattern was chosen to be non-periodic . We will now tackle the periodic case. The general form of a periodic string is $u^s v$, where u is the period of the pattern with $|u| = P$ and $|v| < P$. We let $Q = |v| = m - sP$ which is strictly less than P.

In the perodic case, step 2 of Vishkin's algorithm consists of two parts. In step 2.1 the witness array from step 1 is updated so that for every $2 \leq i \leq P$, $WIT(i) \leq 2P - i$. We will elaborate on the details involved here when we consider the pattern analysis phase later in this chapter. Our sample pattern $(abc)^4 a$ is shown below. For now we will assume that the witness array has the proper format. Note that we only use witness values for $PAT[1,\ldots,2P]$.

	1	2	3	4	5	6	7	8	9	10	11	12	13
PATTERN	a	b	c	a	b	c	a	b	c	a	b	c	a
WIT	0	1	1	0	1	1	-	-	-	-	-	-	-

Our sample text is depicted in figure 4.2(a) along with the $MATCH$ array which has been initialized as before with one modification. Now we enter T's in all locations i in $MATCH$, $1 \leq i \leq n - (2P + 1) + 1$ which corresponds to positions 1 through 18.

Step 2.2 consists of $\lfloor \log P \rfloor$ rounds of duels as before. Since the period size $P = 3$ in our example, only 1 stage will be required. We must calculate the $LEFT$ array where a varies between 1 and $\lceil (24 - 13 + 1)/2^1 \rceil$ or from 1 to 6.

Since each of $LEFT(0,1)$ and $LEFT(0,2)$ contain T's, a duel is required with $j_2 = 1$ and $j_1 = 2$. The situation is depicted on the top of page 70.

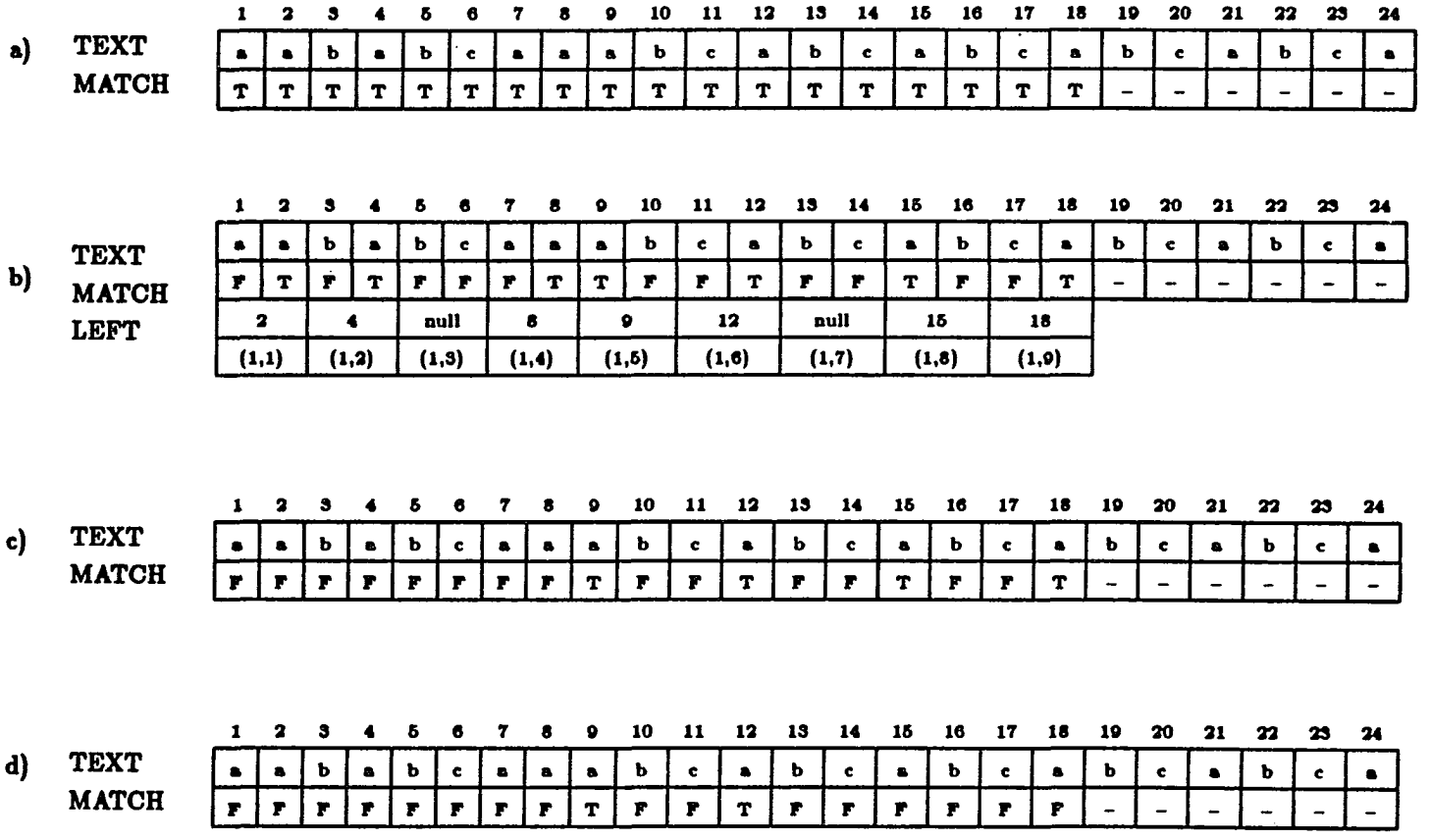
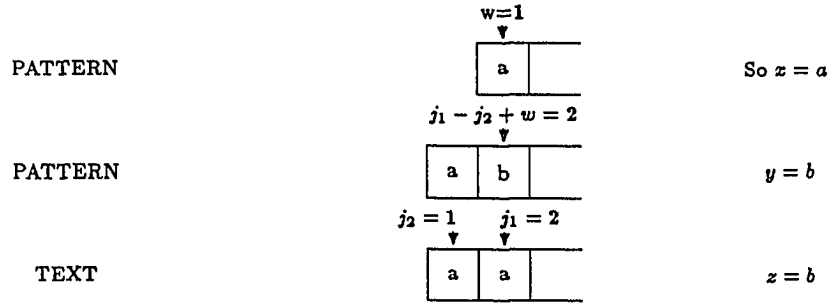


Figure 4.2: Processing of text in Vishkin's Algorithm – a second example.



Since $z \neq y$, $MATCH(j_2)$ is set to F and $LEFT(1,1) \leftarrow j_1$ or 2. By now the format of these duels should be transparent. Eight additional duels are required to obtain the values contained in the $MATCH$ and $LEFT$ arrays shown in figure 4.2(b). The reader may wish to verify several of these values for himself.

Step 3 in the periodic version also consists of two phases. In step 3.1 a character by character check is made at every suspicious index for an occurrence of u^2v which corresponds to the string *abcabca* in our example. The format of the $MATCH$ arrray upon completion of this step is given in figure 4.2(c).

Observe that the T's at locations 2, 4, and 8 have been turned off. There are potential occurrences of the pattern at positions 9, 12, 15, and 18 of the text. Actually the T's in locations 15 and 18 occur too late in the text to

correspond to occurrences of the pattern. However, these entries will prove useful in subsequent calculations. In step 3.2 an attempt is made to find the maximum k such that an occurrence of $u^k v$ begins at i. And if $k \geq s$, then naturally an occurrence of the pattern commences at $TEXT(i)$.

Vishkin employs a balanced binary tree to help in the calculations required in step 3.2. The tree has β leaves where $\beta = n - 2p - Q + 1$. He labels each node in the tree as a pair (x, y) where x is the level in the tree, and y its serial number (counting from left to right) within that level. The leaves of the tree are numbered $(0,1), \ldots, (0, 2^{\lceil \log \beta \rceil})$. A node (x, y) of the tree is the father of two nodes : $(x-1, 2y-1)$ which is its left son and $(x-1, 2y)$ its right son.

The reader may wish to look at figure 4.3 at this time. Indices are drawn outside each node. The entries inside each node will be explained momentarily. An array $LARGEST$ is also used in the calculations. $LARGEST(i)$ will contain the largest index l such that $MATCH(l) = T$ and $PAT[i, \ldots, l-1] = u^{(l-1)/P}$ with $(l-i)/P$ an integer. Note that by adding two to $(l-i)/P$ we obtain that maximum k such that $u^k v$ begins at i.

Vishkin comments that the index l could be obtained in linear time by merely scanning the text from right to left. His parallel approach employs two arrays $A[i, j]$ and $B[i, j]$, their entries correspond to nodes of the tree. In figure 4.3 these entries appear inside the nodes, with A values appearing

to the left and B values on their right. Vishkin's initialization appears below

Initialization

for all i, $1 \leq i \leq 2^{\lceil \log \beta \rceil}$ **pardo**

if $MATCH(i) = T$ and $MATCH(i+P) = F$

// in case the **if** condition is satisfied

the maximun l for i is i itself//

then $A(0,i) \leftarrow i$

else $A(0,i) \leftarrow \infty$

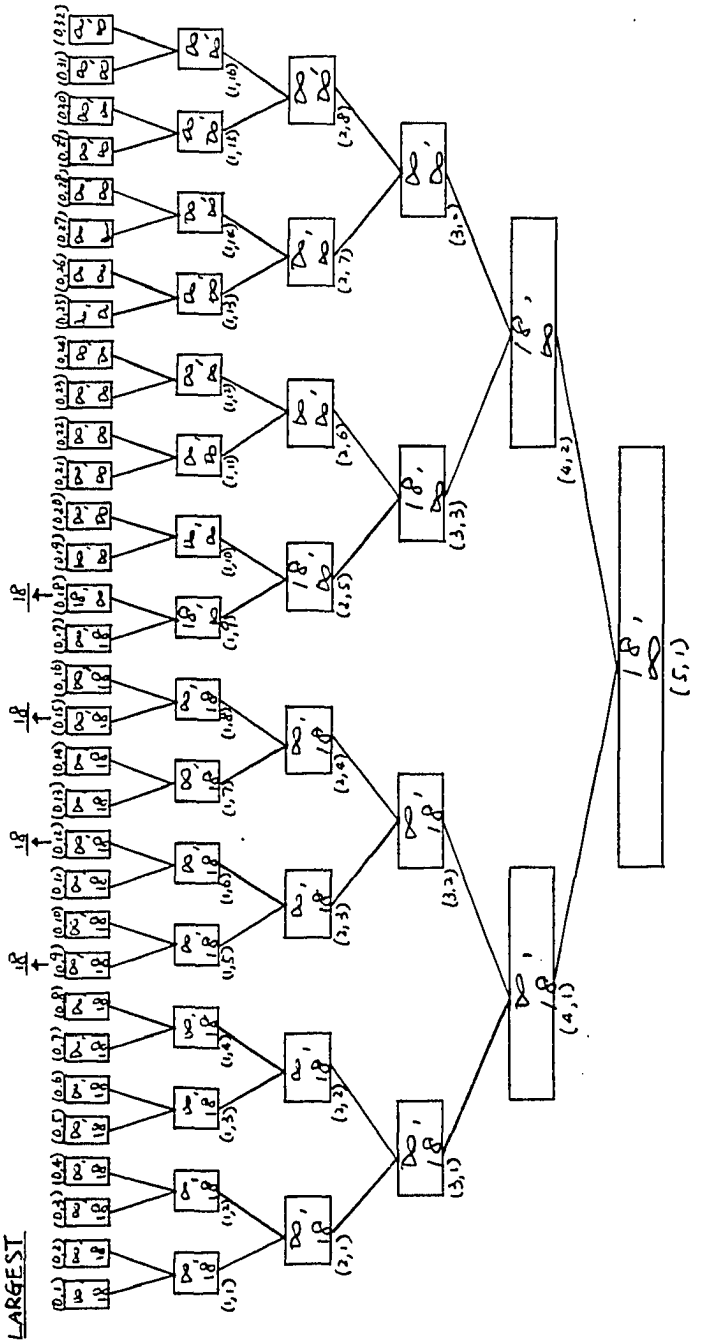


Figure 4.3: Balanced tree used in calculation of LARGEST. Note indices for each node which appear below each box. The sixth node on level 2 is indexed as (2,6). Entries within a node (i,j) correspond to A(i,j) and B(i,j). See text for an explanation of calculations entailed.

A quick glance at the last $MATCH$ array indicates that the if clause above is satisfied only for $i = 18$. By referring to figure 4.3, we observe that $A(0,i) = \infty$ for all leaf nodes except the 18^{th}.

There will be $2\lceil \log \beta \rceil$ or 10 stages in step 3.2. In the first $\lceil \log \beta \rceil$ or 5 stages, we move up the tree one level at a time, starting at the leaves and terminating at the root. The result of these first 5 stages is that each $A(x,y)$ will have the minimum $A(0,i)$ over all its leaf descendents.

stage r, $r \leftarrow 1$ **to** $\lceil \log \beta \rceil$

  for all $i, 1 \leq i \leq 2^{\lceil \log \beta \rceil - r}$ **pardo**

    $A(r,i) \leftarrow min(A(r-1, 2i-1), A(r-1, 2i))$

Perusing figure 4.3 once again, we can see that $A(0,17) = \infty$ and $A(0,18) = 18$. $A(0,18)$ contains the minimum value and hence $A(1,9)$ is set to 18. In subsequent stages, any node that has node (1, 9) as a descendent will obtain an A value of 18 as well.

In the last $\lceil \log \beta \rceil$ or 5 stages, we move down the tree one level at a time, beginning at the root and ending at the leaves. These stages will compute into $B(0,i), 1 \leq i \leq \beta$, the smallest j for which the if condition above is satisfied and $j > i$.

set $B(\lceil \log \beta \rceil, 1) \leftarrow \infty$.

stage $2\lceil \log \beta \rceil + 1 - r, r \leftarrow \lceil \log \beta \rceil$ **downto** 1
  for all $i, 1 \leq i \leq 2^{\lceil \log \beta \rceil - r}$ **pardo**
    $B(r-1, 2i) \leftarrow B(r, i)$
    if $A(r-1, 2i) < \infty$
    then $B(r-1, 2i-1) \leftarrow A(r-1, 2i)$
    else $B(r-1, 2i-1) \leftarrow B(r, i)$;

Referring to figure 4.3 let's trace a few sample calculations. In stage 6 we compute the B values for the 4^{th} level of the tree. First $B(r-1, 2i)$ or $B(4,2)$ is set to ∞. Next it is determined that $A(r-1, 2i)$ or $A(4,2)$ is $< \infty$, actually $A(4,2) = 18$. Therefore $B(r-1, 2i-1)$ or $B(4,1)$ is set to $A(r-1, 2i)$ or $A(4;2)$ which has a value of 18. In stage 8 we are computing B values for the 2^{nd} level of the tree. Consider the calculations entailed for the rightmost nodes in this level, that is $(2,7)$ and $(2, 8)$. First $B(r-1, 2i)$ or $B(2,8)$ is set to the B value of its father, node $(3,4)$. So $B(2,8)$ equals ∞ at this point. Next $A(2,8) = \infty$ so the *else* clause is executed. The result of which is that $B(2,7)$ is set to ∞.

The following routine is used to compute the values of $LARGEST$.

for all $i, 1 \leq i \leq n$, **pardo**
  if $MATCH(i) = T$ and $MATCH(i+P) = F$
  then $LARGEST(i) \leftarrow i$
  else $LARGEST(i) \leftarrow B(0,i)$;

The four entries of interest in the $LARGEST$ array are also depicted in figure 4.3. Each of $LARGEST$ 9, 12, and 15 are set to $B(9), B(12)$ and $B(15)$, respectively, whereas for $i = 18$, the **if** condition is satisfied, indicating that the value for $LARGEST(18)$ is just i itself or 18.

The last phase of the calculations is given below.

for all $i, 1 \leq i \leq \beta$ such that $MATCH(i) = T$ **pardo**
  $k \leftarrow (LARGEST(i) - i)/P + 2$
  if $k < s$
  then $MATCH(i) \leftarrow F$;

For the T in $MATCH(18)$ we have $k \leftarrow (LARGEST(18) - 18)/3 + 2$ which equals $(18 - 18)/3 + 2$ or just 2. Our pattern is $(abc)^4a$ indicating $s = 4$, therefore the T at position 18 is turned off. The T at $MATCH(15)$ is also turned off because k equals $(18 - 15)/3 + 2$ or just 3. However, the

k values at positions 9 and 12 are seen to equal 5 and 4 respectively and hence the T's in these positions remain. The TEXT and $MATCH$ array for the periodic string we have been considering is shown one last time in figure 4.2(c).

One final comment is in order here, Vishkin mentions that it is possible to modify step 3.2 so that only the first $\lceil \log m \rceil$ and last $\lceil \log m \rceil$ stages are required. In this modified version we ascertain whether there is a k, such that $k \geq s$ and $u^k v$ starts at i. There is no attempt made to find the largest such k. The time complexity is thereby reduced to $O(\log m)$.

We have described steps 2 and 3, the text analyses phases of Vishkin's algorithm. It was assumed that an array of witness values for the pattern was available as input to these latter two steps. We now consider the details of the pattern processing which are required in step 1 to yield the desired witness array.

§ *4.4 Processing of Pattern*

Vishkin again employs a set of k-blocks, however in step 1 they refer to the pattern. That is, $\{PAT[1,\ldots,2^k],\ldots,PAT[l\cdot 2^k+1,\ldots,(l+1)2^k],\ldots\}$. Step 1 will consist of $\lfloor \log m \rfloor - 2$ or $\lfloor \log m \rfloor - 3$ iterations, or stages, and a terminal stage. The number of iterations is a function of the pattern length. The algorithm runs as long as there is a $(k+1)$-block, of size 2^{k+1} that serves as a buffer between $\lfloor m/2 \rfloor + 1$ and the end of the pattern.

When $m \geq 3 \cdot 2^{\lfloor \log m \rfloor} - 1, \lfloor \log m \rfloor - 2$ stages are performed. And when $m < 3 \cdot 2^{\lfloor \log m \rfloor} - 1$, only $\lfloor \log m \rfloor - 3$ stages.

These properties will be satisfied following stage k. They are :

- the k-certainty property. For all j between 1 and 2^k, $WIT(j) = 0$ if and only if an occurrence of $PAT[1, \ldots, 2^{k+1} - j + 1]$ starts at $PAT[j]$. Suppose that $PAT[1, \ldots, 2^{k+1}]$ contains the whole pattern, then $WIT(1, \ldots, 2^k)$ would have its final values.

- the k-sparsity property. (here we are referring to the pattern). If $WIT[2, \ldots, 2^k]$ does not contain any zero entries, then WIT of each k-block will contain at most one zero. In other words, each of $WIT[1, \ldots 2^{k+1}], \ldots WIT[l2^k + 1, \ldots (l+1)2^{k+1}], \ldots$ will have at most one zero.

- the k-lookahead property. $WIT(i) \leq 2^{k+1}$ for each index i of the pattern.

Vishkin comments that satisfying the k-certainty and k-sparsity properties seems like an intuitive goal, but that satisfying the k-lookahead property seems perhaps counter-intuitive. He proves in his paper (his Lemma 1) that indeed the k-lookahead property is satisfied. His proof proceeds by induction on k.

Vishkin provides a flowchart description for step 1. Refer to figure 4.4. We assume that we are in stage $k + 1$ of step 1. After stage k, we will

be at the arrow leading to either **Box 2** or **Box 4**. k-certainty is satisfied in either case. When proceeding toward **Box 2**, "k-sparsity" is satisfied. However if we were in **Box 4** in stage k and remain in **Box 4** for stage $k+1$, k-sparsity need not be satisfied. In this situation, the algorithm is in the periodic mode.

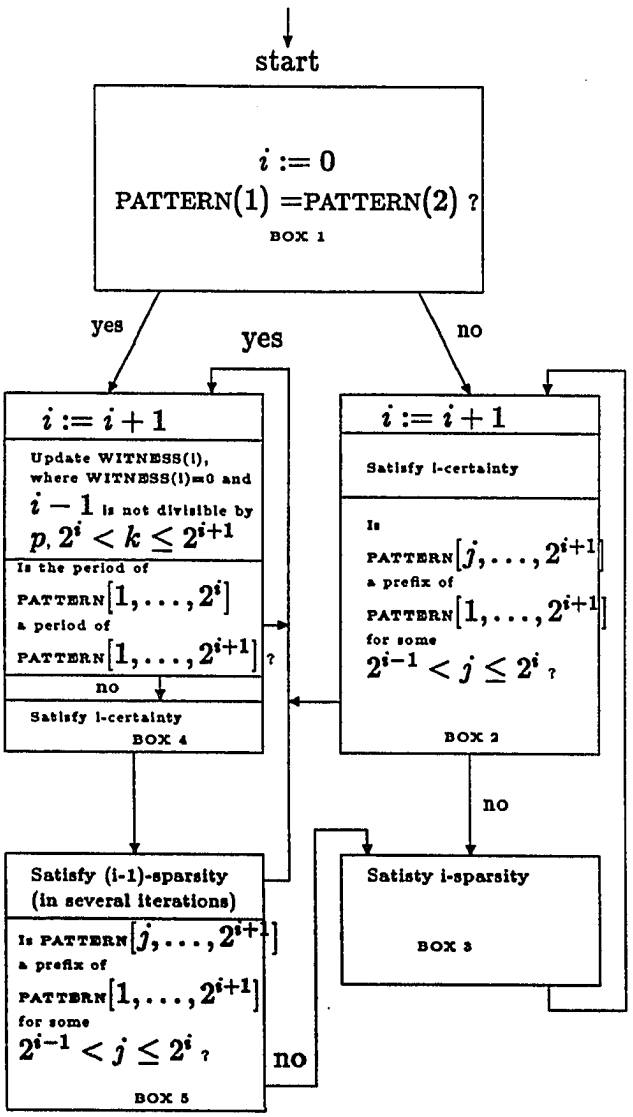


Figure 4.4: Step 1 of Vishkin's Algorithm.

Once again a series of left arrays are employed. $LEFT(k, a)$ contains the location of the leftmost zero in WIT of k-block a, where $1 \leq a \leq \lceil m/2^k \rceil$ or a null entry, indicating that $PERIODICITY(k)$ will contain the period size.

The initialization takes phace in **Box 1.** Vishkin's pseudocode appears below.

Box 1 (Start)

for all $j, 1 \leq j \leq m$ **pardo**
    $WIT(j) \leftarrow 0$;
if $PAT(1) \neq PAT(2)$
then start stage 1 at **Box 2**
else start stage 1 at **Box 4** // enter a periodic mode //

Our first sample pattern was *ababaaab*. However, to make the explanation detailed enough we consider a pattern consisting of three concatenated copies of this string. This new pattern along with the contents of the WIT and $LEFT(0, a)$ arrays as they would appear after this initialization are shown in the figure 4.5(a).

Since $PAT(1) \neq PAT(2), i.e., a \neq b$, we enter stage 1 at **Box 2.**

Box 2

When we enter **Box 2**, k-certainty and k-sparsity are satisfied. Note at stage 1, the latter property is satisfied in a trivial sense as block size equals one at this time. Here, a check is made for potential periodicity. If it is found, stage $k+2$ begins at **Box 4**, the periodic mode, else we progress to **Box 3.** The pseudocode for **Box 2** follows.

if $LEFT(k,2) \neq$ *"null"*

// i.e., Is there a zero in k-block number 2 of the WIT array, k-sparsity being satisfied implies there can be at most one zero //

then let $x = LEFT(k,2)$

for all $j, 1 \leq j \leq 2^{k+2} - x + 1$ **pardo**

if $PAT(j) \neq PAT(x-1+j)$

then $WIT(x) \leftarrow j$

// It is possible that the if condition holds for several j's, in which case simultaneous WRITEs into $WIT(x)$ would occur //

if $WIT(x) = 0$ //the condition did not hold //

then $PERIODICITY(k+1) \leftarrow x - 1$

// start stage $k+2$ at **Box 4** //

Proceed to **Box 3** (i.e., $WIT[2,\ldots,2^{k+1}]$ has no zeros)

In our sample pattern, there is a zero in the second k-block. Since $k = 0$ we are merely observing that $WIT(2) = 0$. We set x to $LEFT(0,2)$ or 2. Then for all $1 \leq j \leq 2^{0+2} - 2 + 1$, i.e., for all $1 \leq j \leq 3$ we perform the following comparisons in parallel:

$$PAT(1) \neq PAT(2), \quad a \neq b$$
$$PAT(2) \neq PAT(3), \quad b \neq a$$
$$PAT(3) \neq PAT(4), \quad a \neq b$$

The inequality was satisfied in all three comparisons, hence simultaneous WRITEs into $WIT(2)$ would take place. We assume the lowest value for j would appear as the value for $WIT(2)$, i.e., $WIT(2) = 1$. At this point, the second k-block contains no zeros. We proceed to **Box 3.**

Box 3

As we enter this box, k-sparsity and $(k+1)$-certainty are satisfied. For every $2 \leq j \leq 2^{k+1}, WIT(j) \neq 0$. The following procedure is performed in parallel for all $(k+1)$-blocks. We are attempting to satisfy $(k+1)$-sparsity. We let a be an integer such that: $2 \leq a \leq \lceil m/2^{k+1} \rceil$. As with the text analysis, $(k+1)$-block a is the union of two k-blocks : $2a$ and $2a-1$.

if $LEFT(k, 2a) =$ "*null*"
then $LEFT(k+1, a) \leftarrow LEFT(k, 2a-1)$
else if $LEFT(k, 2a-1) =$ "*null*"
then $LEFT(k+1), a) \leftarrow LEFT(k, 2a)$
else see below

Since k-sparsity is satisfied when we enter **Box 3**, there will be at most one index j_1 in k-block $2a$ and at most one index j_2 in k-block $2a-1$ with their respective WIT values equal to zero. Should both of these indices, j_1 and j_2 exist, a duel will be employed to eliminate one of the zeros. We let $w = WIT(j_1 - j_2 + 1)$, $x = PAT(W)$, $y = PAT(j_1 - j_2 + w)$ and $z = PAT(j_1 - 1 + w)$.

Vishkin remarks that we may ignore the case when there is a reference to an index of the pattern which is $> m$, i.e., when $j_1 - 1 + w > m$. In

this situation, it is assumed that $PAT(j_1 - 1 + w)$ matches any character of the pattern. He assumes that the k-lookahead property guarantees that this will not alter the algorithm's correctness.

Z will be employed to eliminate one or both of the zeros at j_1 and j_2.

if $z \neq y$
then $WIT(j_2) \leftarrow j_1 - j_2 + w$
if $z \neq x$
then $WIT(j_1) \leftarrow w$

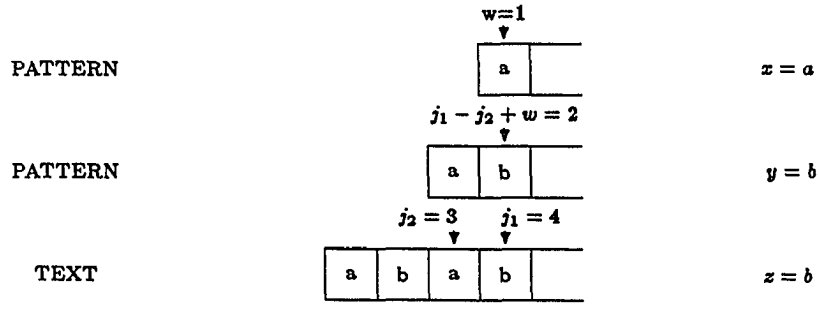
if $WIT(j_2) = 0$
then $LEFT(k+1, a) \leftarrow j_2$
else **if** $WIT(j_1) = 0$
then $LEFT(k+1, a) \leftarrow j_1$
else $LEFT(k+1, a) \leftarrow null$

We are in stage 1 and are therefore attempting to satisfy 1-sparsity, i.e., every 2^1 block of pattern should contain at most a single zero. The contents of the WIT and $LEFT$ arrays as they appear at this juncture are shown in figure 4.5(b).

A quick scan of the WIT array makes it apparent that eleven duels in parallel will be required to decrease the number of zeros. We provide the

details of the first of these duels below.

In $(k+1)$-block number two, we observe zeros at each of positions $j_2 = 3$ and $j_1 = 4$. We let $w = WIT(j_1 - j_2 + 1)$, i.e., $w = WIT(2) = 1$. $x = PAT(w)$, indicating $x = PAT(1) = a$, $y = PAT(j_1 - j_2 + w)$ which is $PAT(2) = b$. And $z = PAT(j_1 - 1 + w)$, i.e., $z = PAT(4)$ which is a "b". The ensuing duel is depicted below.



We have that $z \neq x$, therefore it is impossible for the suffix commencing at position $j_1 = 4$ to be a prefix of the pattern. Hence $WIT(4)$ is set to 1. Subsequently, $LEFT(1,2)$ is set to $j_2 = 3$. Ten other duels are required to ensure $k+1$ or 1-sparsity. The reader may wish to carry out a few of these on his own. The updated WIT array as it appears after stage 1 is shown in figure 4.5(c) with the $LEFT(1,a)$ array.

We begin stage 2 in **Box 2.** We first check if $LEFT(k,2) \neq$"null". Indeed $LEFT(1,2)$ contains a 3 indicating $WIT(3) = 0$. We set x to this value, i.e., $x \leftarrow 3$. Then for all j between 1 and $2^{k+2} - x + 1$, i.e., $1 \leq j \leq 6$ we perform the following six comparisons in parallel:

if $PAT(j) \neq PAT(x - 1 + j)$ i.e.,

$PAT(1) \neq PAT(3)$ // actually this comparison is superfluous (refer to stage 1 for details)//

$PAT(2) \neq PAT(4)$, $b \neq b$, *no*

$PAT(3) \neq PAT(5)$, $a \neq a$, *no*

$PAT(4) \neq PAT(6)$, $b \neq a$, *yes*

$PAT(5) \neq PAT(7)$, $a \neq a$, *no*

$PAT(6) \neq PAT(8)$, $a \neq b$, *yes*

For two comparisons, the **if** clause was satisfied indicating that two processors will attempt to update $WIT(3)$ simultaneously, with values of 4 and 6 respectively. We assume the lower value wins out. At this point $WIT[2,\ldots,4]$ contains no zeros and once again we proceed to **Box 3.**

We study the $LEFT(1,a)$ for a ranging from 3 to 12. It is evident that to obtain the $LEFT(2,a)$ entries, five duels performed in parallel will be required. The updated WIT and $LEFT$ arrays as they appear after stage

2 are shown in figure 4.5(d).

Our pattern length m equals 24 in this first example. Since $24 \geq 3 \cdot 2^{\lfloor \log 24 \rfloor - 1}$, actually $24 = 3 \cdot 2^{4-1}$, $\lfloor \log 24 \rfloor - 2$ or just 2 stages need to be performed before executing the terminal stage. Vishkin refers to this as case 1. Notice that we did not yet consider the periodic mode of Vishkin's algorithm, i.e., boxes 4 and 5 in figure 4.4. When we analyze a second pattern example, this part of the algorithm will be explained.

Actually case 1 breaks into two subcases.

Case 1.1 $PAT[1,\ldots,2^{\lfloor \log m \rfloor - 1}]$ does not have a period of size $\leq 2^{\lfloor \log m \rfloor - 2} - 1$. In our example $PAT[1,\ldots,2^3]$ does not have a period size ≤ 3. This implies that had there been a stage $\lfloor \log m \rfloor - 1$, i.e., stage 3, it would have begun in **Box 2**. We know that $WIT[2,\ldots,2^{\lfloor \log m \rfloor - 2}]$ does not contain a zero, this refers to $WIT[2,3,4]$ in our example. There is at most one zero in each of $(\lfloor \log m \rfloor - 2)$-blocks 2, 3, and 4. In our first pattern, we have exactly one zero in each of 2-blocks 2, 3 and 4. For each of these zeros, at locations 7, 9, and 15 check in a character by character fashion if they indicate periods of the pattern. If yes, then of course the zeros may remain, otherwise update the WIT array using perhaps simultaneous WRITEs into the same positions.

The pattern and WIT array are shown in figure 4.5(e) as they appear when step 1 is complete. The $LEFT$ array is no longer needed and is

therefore not included.

We are only interested in entries of WIT which are less than or equal to $\lfloor m/2 \rfloor + 1$, or ≤ 13 in this pattern. Therefore the only zero of interest is in position 9, indicating a period size of 8. We are done.

In case 1.2 $PAT[1,\ldots,2^{\lfloor \log m \rfloor - 1}]$ has a period whose size P is less than or equal to $2^{\lfloor \log m \rfloor - 2} - 1$. We shall defer an explanation of this case until after we have considered a second sample pattern. Our second example should help to elucidate the periodic mode of Vishkin's algorithm.

We now look at step 1 when the pattern is $(ab)^4a(ab)a(ab)a(ab)^2$ $a^2(ab)^2a^4a(ab)^4a(ab)^2a^2(ab)^2$. Since once again the pattern length $m \geq 3 \cdot 2^{\lfloor \log m \rfloor - 1}$, $\lfloor \log m \rfloor - 2$ or in this case 3 stages are required before reaching the terminal stage. We display the WIT and $LEFT$ arrays as they would appear after initialization in figure 4.6(a).

Referring once again to figure 4.6(a) we observe that $PAT(1) \neq PAT(2)$, i.e., the a in position 1 of the pattern is not equal to the b in position 2. We therefore begin stage 1 at **Box 2.** As in the prior example, we will check for potential periodicity. The reader may wish to refer to the pseudocode provided earlier for this Box. We observe in this example that there is indeed a non-null entry in $LEFT(0,2)$. Hence x is set to the contents thereof which is 2. Next for all $1 \leq j \leq 3$ we compare $PAT(j)$ to $PAT(x - 1 + j)$. Each comparison satisfies the inequality condition and therefore $WIT(2)$ is updated. Once more we assume that the lowest

value for j which satisfies the inequality is written into location 2 of WIT, resulting in $WIT(2) = 1$. We next enter **Box 3.**

In **Box 3** we attempt to satisfy $(k+1)$ or 1-sparsity. Scanning the contents of $WIT(3)$ through $WIT(48)$ we observe that each entry is a zero. Satisfying 1-sparsity will require 23 duels in parallel. The contents of the WIT and $LEFT$ arrays after these duels are completed is depicted in figure 4.6(b). The reader is invited to carry out a few of these duels on his own to verify the entries provided. Also, we should remark that since j_1 was always equal to $j_2 + 1$ in each duel, w had a value of 1 in each case, guaranteeing that $j_1 - 1 + w$ was never greater than m. Thereby ensuring that 1-sparsity would be satisfied for every 1-block. Stage 1 is complete, we proceed to **Box 2** to begin stage 2.

In **Box 2** we check for suspected periodicity. We do so as usual by checking the second entry in the left array. $LEFT(1,2)$ contains a 3, we therefore set x to 3. Then for all $1 \leq j \leq 2^{k+2} - x + 1$, or for all j between 1 and 6 we compare $PAT(j)$ with $PAT(x - 1 + j)$. It is readily seen that all of these character comparisons are successful, therefore, we cannot update $WIT(3)$. Instead we set $PERIODICITY(2) \leftarrow x - 1$ or 2. We start stage 3 in **Box 4,** the periodic case. The WIT and $LEFT$ arrays have not been updated and hence figure 4.6(b) depicts the contents of these arrays after stage 2 as well. Notice that $k = 2$-sparsity is *not* satisfied at this point.

As commented earlier, stage 3 begins in **Box 4.** 2-certainty and 1-

sparsity are satisfied. Vishkin advices us to pick indices j of the first 3-block such that $WIT(j) = 0$ and $j - 1$ is not divisible by P (or 2 in this example). A quick glance at figure 4.6(b) indicates that in our WIT array no such j values exist. In the case that such an index exists, we are to choose an index j, with $j - P < i < j$, and $i - 1$ divisible by P and perform what Vishkin refers to as a "one way" duel between i and j in which only assignments into $WIT(j)$ can be made. We choose $i = [(j-1)/P] \cdot P + 1$, then

for all $j, 2^k < j \leq 2^{k+1}$
  if $WIT(j) = 0$ **and** $j \bmod P \neq 1$
  then (let $w = WIT(j \bmod P)$)
    if $PAT(j - 1 + w) \neq PAT(w)$
    then $WIT(j) \leftarrow w$

Vishkin notes that if P is not a period of $PAT[1, \ldots, 2^{k+2}]$, then for at most four indices j that satisfied the first if condition $WIT(j)$ remains 0. For a proof one may consult his paper.

We now check if the periodicity continues until 2^{k+2}. Since this is stage $k + 1$, or stage 3, we are checking up to 2^4 or position 16. If yes, return

to **Box 4** for the next stage (or perform a terminal stage case 1.2 , to be described shortly). If the periodicity does not continue proceed to **Box 5.**

for all $j, 2^{k+1} < j \leq 2^{k+2}$ **pardo**
  if $PAT(j) \neq PAT(j \bmod P)$
  then $WIT(P+1) \leftarrow j - P$
    // simultaneous WRITEs are possible and the lowest $j - P$ will be assigned //
  if $WIT(P+1) = 0$
  then start stage $k+2$ at **Box 4**
  else for all $j, 2 \leq j \leq (2^{k+1} - 1)/P$ **pardo**
    $WIT(jP+1) \leftarrow WIT(P+1-(j-1)P$

So for our example, in stage 3, j will assume values between 2^3 and 2^4 or from 8 to 16. Observe that $PAT(11) \neq PAT(11 \bmod 2)$ or $PAT(1)$, i.e., the b in the eleventh position does not match the first pattern character which is an a. We reset $WIT(P+1)$, i.e., $WIT(3)$ to $j - P$ which is 9. We no longer suspect the string ab of being the period of pattern. Next for all j between 2 and $(2^3 - 1)/2$ i.e., for $j = 2$ and 3. $WIT(jP+1) \leftarrow WIT(P+1) - (j-1)P$, i.e., $WIT(5)$ is set to $WIT(3) - (2-1)P$ or $9 - 2$

which is 7. And $WIT(7)$ is set to 5. Notice that it is the same character, namely the b in position 11 of the pattern which is contradicting periods of size jP , for each value of j.

for each $i, 2^k < i \leq 2^{k+1}$ such that $WIT(i) = 0$ **pardo**
// there are at most four such i's //
for all $j, 1 \leq j \leq 2^{k+2} - i + 1$ **pardo**
if $PAT(i) \neq PAT(i - 1 + j)$
then $WIT(i) \leftarrow j$
// simultaneous WRITEs are possible //
if $WIT(i) = 0$
// i.e., the condition did not hold for any j //
then $PERIODICITY(k + 1) \leftarrow j - 1$;
proceed to **Box 5**

By referring to figure 4.6(c) we note that these are no indices i between 4 and 8 with $WIT(i) = 0$, hence we may just go to **Box 5.**

Box 5. $k + 1$ certainty which is 3-certainty in our example is satisfied. $WIT[2, \ldots, 2^2]$ has no zeros. k-sparsity or 1-sparsity in this case is satisfied. We need to satisfy $k+1$ = 3-sparsity. In **Box 5** we satisfy k = 2-sparsity and

then proceed to **Box 3** where $k+1$ = 3-sparsity would be satisfied. So $k-k_1$ or 1 iteration is required in **Box 5**. The approach here is similar to that of **Box 3**. Whenever $LEFT$ boxes $2a$ and $2a-1$ both contain a non-null entry a duel will be required. Figure 4.6(d) shows the contents of the $LEFT(2,a)$ array after 2-sparsity has been satisfied. Every $LEFT(1,a)$ entry for $5 \leq a \leq 24$ contained a non-null entry (see figure 4.6(c)). Therefore, 10 duels in parallel (performed in the usual manner) were required to obtain 2-sparsity. Notice though that two zeros are present in the last 2-block of the witness array. In this rightmost duel $j_2 = 45$, $j_1 = 47$ and $w = 9$, therefore, $z = PAT(55)$ which does not exist.We treat z like a wild card which matches any character it is compared with. Neither $WIT(j_2)$ nor $WIT(j_1)$ is reset and so we do not have 2-sparsity in this last block. As only the first $\lfloor m/2 \rfloor + 1$ entries of WIT are used in the text processing phase (Steps 2 and 3), this will not affect the outcome of the algorithm.

We now proceed to **Box 3**, where 3-sparsity will be established. Since there are non-null entries in $LEFT(2,a)$ for $3 \leq a \leq 12$, five duels will be performed. The WIT and $LEFT$ array once **Box 3** is complete are shown in figure 4.6(e). At this point stage 3 is complete. We next perform a terminal stage.

Since our pattern length is 48, case 1 holds. Furthermore we have that $PAT[1,\ldots,2^{\lfloor \log m \rfloor} - 1]$ i.e., $PAT[1,\ldots,16]$ does not have a period of size $\leq 2^{\lfloor \log m \rfloor - 2} - 1$, i.e., less than or equal to 7, hence subcase 1.1 applies here.

If there had been a fourth stage (i.e., stage $\lfloor \log m \rfloor - 1$) it would have begun in **Box 2**. $WIT[2,\ldots,2^{\lfloor \log m \rfloor} - 2]$, i.e., $WIT[2,\ldots,8]$ does not contain a zero. Each one of 3-blocks 2, 3, and 4 may have at most a single zero each. In our example they are located in positions 16, 24, and 30 of the witness array. For each of these positions check in a character by character comparison if they stand for a period of the pattern. If not, then update the witness array, using simultaneous WRITEs as indicated. In checking the 0 at $WIT(16)$, we discover that $PAT(21) \neq PAT(6)$. For the 0 at $WIT(24)$, we have that $PAT(27) \neq PAT(4)$. And finally on testing the 0 at $WIT(30)$, we obtain a mismatch between $PAT(42)$ and $PAT(13)$. So that $WIT(16)$,$WIT(24)$, and $WIT(30)$ are updated to 6, 4, and 13 respectively. Step 1 of the algorithm is finished (see figure 4.6(f)). The first $\lfloor m/2 \rfloor + 1$ or first 25 entries of the witness array would be employed in the text processing steps of the algorithm.

We mentioned earlier that the terminal stage for case 1.2 was invoked when $m \geq 3 \cdot 2^{\lfloor \log m \rfloor - 1}$, and had there been a stage $\lfloor \log m \rfloor - 1$ it would have begun at **Box 4**. This latter condition of course indicates that the algorithm is in the periodic mode. Assume that our pattern is $(ab)^{24}$. Clearly ab is the period of the entire string. The terminal stage in this scenario essentially mirrors the explanation provided in **Box 4** and **Box 5**, and so we spare the reader the details. For the sake of completeness we mention that case 2 for the terminal stage occurs when the pattern length

$< 3 \cdot 2^{\lfloor \log m \rfloor - 1}$. There are two subcases here as well depending on whether the pattern is periodic or not. Subcases are handled in the same manner as subcases 1.1 and 1.2 described above.

One final note is in order. We mentioned in the explanation of the text analysis steps, that in the case where the pattern is periodic, the entries in the witness array for $2 \leq i \leq P$ should be $\leq 2P - i$. Vishkin comments that since the pattern has a period of length P, this implies that $w - P$ is also a witness for i. We let $c > 0$ be any integer such that $w - cP > 0$. We may then assign $w - cP$ into $WIT(i)$. Vishkin just selects c equal to $\lfloor (w - 2P + i)/P \rfloor$ and then assigns $w - cP$ into $WIT(i)$, yielding $WIT(i) \leq 2P - 1$ as desired. In the beginning of this chapter we employed WIT arrays for the patterns *ababaaab* and $(abc)^4a$. The reader may wish to test his understanding of the algorithm by deriving the WIT array in each case.

Vishkin comments on the complexity of his algorithm. Stage k of step 2 (text analysis phase) requires $O(n/2^n)$ operations and constant time. Step 2 may be performed a maximum of $\log m$ times and hence requires $O(n)$ operations and $O(\log m)$ time. When the pattern is not periodic, step 3 requires $O(n)$ operations and constant time. In the periodic case, where a balanced tree was employed, $O(n)$ operations and $O(\log m)$ time is needed. Therefore steps 2 and 3 required in total, $O(n)$ operations and $O(\log m)$ time when $n/\log m$ processors are used. The pattern analysis carried out in

step 1 entails $O(m)$ operations and may be accomplished in $O(\log m)$ time when $m/\log m$ processors are available. Therefore the entire algorithm has a parallel running time of $O(\log m)$ assuming $n/\log m$ processors are available.

a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b
WIT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LEFT(0,a)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

b)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b
WIT	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LEFT(0,a)	1	-	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

c)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b
WIT	0	1	0	1	2	0	0	1	0	1	0	1	2	0	0	1	0	1	0	1	2	0	0	1
LEFT(1,a)	1		3		6		7		9		11		14		15		17		19		22		23	
	(1,1)		(1,2)		(1,3)		(1,4)		(1,5)		(1,6)		(1,7)		(1,8)		(1,9)		(1,10)		(1,11)		(1,12)	

Figure 4.5: Processing of pattern in Vishkin's Algorithm.

d)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b
WIT	0	1	4	1	2	2	0	1	0	1	4	1	2	2	0	1	0	1	4	1	2	2	0	1
LEFT(2,a)	1				7				9				15				17				23			
	(2,1)				(2,2)				(2,3)				(2,4)				(2,5)				(2,6)			

e)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b	a	b	a	b	a	a	a	b
WIT	0	1	4	1	2	2	6	1	0	1	4	1	2	2	6	1	0	1	4	1	2	2	0	1

Figure 4.5: Step one of Vishkin's Algorithm continued.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	b	a	b	a	a	b	a	a	b	a	a	b	a	b	a	a	a	b	a
WIT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LEFT(0,a)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
b	a	a	a	a	a	b	a	b	a	b	a	b	a	a	b	a	b	a	a	a	b	a	b
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48

a) WIT and LEFT arrays after initialization.

Figure 4.6: WIT and LEFT arrays for pattern after each phase of step 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	b	a	b	a	a	b	a	a	b	a	a	b	a	b	a	a	a	b	a
WIT	0	1	0	1	0	1	0	1	2	0	2	0	0	1	2	0	2	0	2	0	2	0	2	0
LEFT(1,a)	1		3		5		7		10		12		13		16		18		20		22		24	
	(1,1)		(1,2)		(1,3)		(1,4)		(1,5)		(1,6)		(1,7)		(1,8)		(1,9)		(1,10)		(1,11)		(1,12)	

	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
	b	a	a	a	a	a	b	a	b	a	b	a	b	a	a	b	a	b	a	a	a	b	a	b
	2	0	2	0	2	0	2	0	2	0	2	0	2	0	0	1	0	1	2	0	0	1	0	1
	26		28		30		32		34		36		38		39		41		44		45		47	
	(1,13)		(1,14)		(1,15)		(1,16)		(1,17)		(1,18)		(1,19)		(1,20)		(1,21)		(1,22)		(1,23)		(1,24)	

b) WIT and LEFT arrays after completing of stage 1.
Note these arrays will look this way after stage 2 as well.

Figure 4.6: WIT and LEFT arrays for pattern after each phase of step 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	b	a	b	a	a	b	a	a	b	a	a	b	a	b	a	a	a	b	a
WIT	0	1	4	1	6	2	0	1	0	1	4	1	6	2	0	1	0	1	4	1	0	2	0	1
LEFT(1,a)	1		null		null		null		10		12		13		16		18		20		22		24	
	(1,1)		(1,2)		(1,3)		(1,4)		(1,5)		(1,6)		(1,7)		(1,8)		(1,9)		(1,10)		(1,11)		(1,12)	

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
b	a	a	a	a	a	b	a	b	a	b	a	b	a	a	b	a	b	a	a	a	b	a	b
0	1	4	1	6	2	0	1	0	1	4	1	6	2	0	1	0	1	4	1	0	2	0	1
26		28		30		32		34		36		38		39		41		44		45		47	
(1,13)		(1,14)		(1,15)		(1,16)		(1,17)		(1,18)		(1,19)		(1,20)		(1,21)		(1,22)		(1,23)		(1,24)	

c) WIT and LEFT arrays after completing Box4 of Stage 3.

Figure 4.6: Processing of pattern in Vishkin's Algorithm continued.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	b	a	b	a	a	b	a	a	b	a	a	b	a	b	a	a	a	b	a
WIT	0	1	4	1	6	2	0	1	0	1	4	1	6	2	0	1	0	1	4	1	0	2	0	1
LEFT(2,a)	1				null				12				16				20				24			
	(2,1)				(2,2)				(2,3)				(2,4)				(2,5)				(2,6)			

	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
	b	a	a	a	a	a	b	a	b	a	b	a	b	a	a	b	a	b	a	a	a	b	a	b
	0	1	4	1	6	2	0	1	0	1	4	1	6	2	0	1	0	1	4	1	0	2	0	1
	28				30				36				39				44				45			
	(2,7)				(2,8)				(2,9)				(2,10)				(2,11)				(2,12)			

d) WIT and LEFT arrays after completing Box 5 of stage 3.

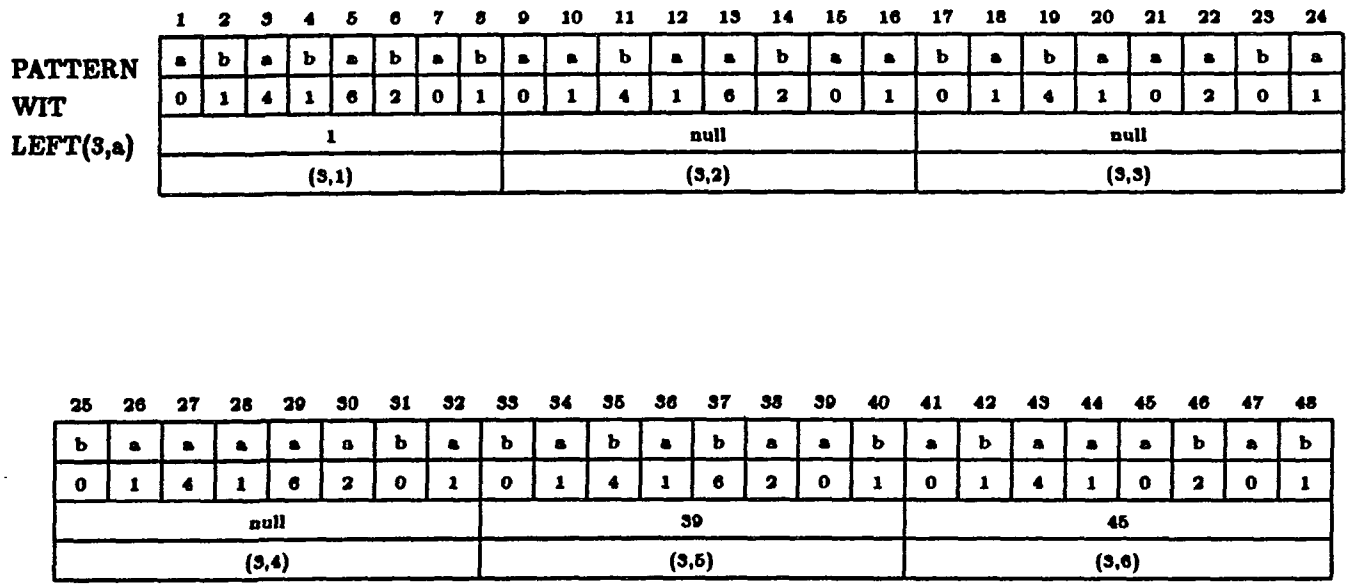
Figure 4.6: Processing of pattern in Vishkin's Algorithm continued.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
PATTERN	a	b	a	b	a	b	a	b	a	a	b	a	a	b	a	a	b	a	b	a	a	a	b	a
WIT	0	1	4	1	6	2	0	1	0	1	4	1	6	2	0	1	0	1	4	1	0	2	0	1
LEFT(3,a)	1								16								24							
	(3,1)								(3,2)								(3,3)							

	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
	b	a	a	a	a	a	b	a	b	a	b	a	b	a	a	b	a	b	a	a	a	b	a	b
	0	1	4	1	6	2	0	1	0	1	4	1	6	2	0	1	0	1	4	1	0	2	0	1
	30								39								45							
	(3,4)								(3,5)								(3,6)							

e) WIT and LEFT arrays after completing Box3 of Stage 3.

Figure 4.6: Processing of pattern in Vishkin's Algorithm continued.



f) WIT and LEFT arrays once stage 1 is complete.

Figure 4.6: Processing of pattern in Vishkin's Algorithm continued.

CHAPTER 5

RABIN KARP ALGORITHM

§5.1 *Algorithms for string matching*

Rabin and Karp [*KARP*81] have suggested an approach to string matching which is based on hashing. Essentially a hash code Φ_p, is associated with the pattern, (called a fingerprint by the authors). We assume that the pattern length is m, and that n is the length of the text string. This same function Φ_p is then applied to each segment of length m of the text. If the fingerprint for some "m-segment" of the text equals Φ_p evaluated for the pattern, then there is a good chance that a match has been found, (recall that with hashing there is always a chance of collisions). If we are unwilling to accept a small probability of error, then whenever a suspected match is discovered, a character by character comparison between the pattern and text segment is indicated.

For this strategy to be effective timewise, it is essential that fingerprints be easily updatable. That is, given Φ_p for m-segment i of the text

$(t_i, t_{i+1}, \ldots, t_{i+m-1})$ it should be an easy chore to compute the fingerprint for m-segment $i+1, (t_{i+1}, t_{i+2}, \ldots, t_{i+m})$. Rabin and Karp's approach treats strings as binary integers and relies upon various mod functions to enlist "pipelining" into their calculation of fingerprints for text segments. They also show how their basic algorithm can be modified to handle 2-dimensional pattern matching of rectangular and of irregular patterns as well. They also derive a real time version for their basic algorithm.

We assume that the pattern $P = p_1 p_2 \ldots p_m$ and text $T = t_1 t_2 \ldots t_n$ each consist of binary symbols, i.e., the alphabet V equals $\{0, 1\}$. This latter restriction may be eased if inconvenient. Let $T(i) = t_i t_{i+1} \ldots t_{i+m-1}$, then whenever $P = T(i)$ there is a match, Rabin and Karp define:

$$a = p_1 2^{m-1} + \ldots + p_m$$

$$a(i) = t_i 2^{m-1} + \ldots + t_{i+m-1}$$

where $1 \leq i \leq n - m + 1$. The condition for a match is that $a = a(i)$. Some additional notation is required. Let $res(b, c)$ represent the residue of b when divided by c. And we let p be a randomly chosen prime in the range $[1, mn^2]$. As shorthand we denote $res(b, p)$ as $\bar{b}$, and $\bar{\sigma}$ will represent the operation σ mod p for $\sigma = +, -, \cdot$. Initially

$$\bar{a} = \overline{\overline{\overline{(p_1 \cdot \overline{2} + p_2)} \cdot 2} + 3 \cdots},$$

and $\overline{a(o)}$ are computed.

After the i^{th} step in the calculations, where $1 \leq i \leq n-m+1$, $\bar{a}$ and $\overline{a(i)}$ have been computed. When $\bar{a} \neq \overline{a(i)}$ then $\overline{a(i+1)}$ is obtained as follows:

$$\overline{a(i+1)} = \overline{\overline{a(i)} - \overline{2^{n-1}} \cdot t_i \cdot 2 + t_{i+m}}$$

and $\bar{a}$ is compared to $\overline{a(i+1)}$.

If $\bar{a} = \overline{a(i)}$ for some i then a bit by bit comparison between the pattern and $T(i)$ is made. If there is a match stop and return i, else a new random number $p < mn^2$ is chosen and the search through the text recommences at position $i+1$ of the text. Auxiliary storage is required for $p, \bar{a}, \overline{a(i)}, \overline{2^{n-1}}$ and i, where i is our index into the text string.

Rabin and Karp establish a general framework for the class of string matching problems. Particular instances of string matching are conveniently expressible within their framework. They let R be an index set of cardinality t. For each $r \in R, P(r)$ and $T(r)$ are strings in $\{0,1\}^m$. The general string-matching problem is to find an index $r \in R$, if one exists, such that $P(r) = T(r)$. In the straightforward string-matching problem that we have been considering, the pattern $P = p_1 p_2 \ldots p_m$ and the text $T = t_1 t_2 \ldots t_n$. Then $t = n - m + 1, R = \{1, 2, \ldots, n - m + 1\}$, $P(r) = P$ for all r, and $T(r) = t_r t_{r+1} \ldots t_{r-m+1}$. S is a finite set and, for $p \in S, \Phi_p$ is a function from $\{0,1\}^m$ into some range D_p. In the following algorithm,

$a_p(r)$ and $b_p(r)$ will be shorthand for $\Phi_p(P(r))$ and $\Phi_p(T(r))$, respectively. Let α denote the first element of R and ω, the last. For all $r \in R - \{\omega\}, r$ will denote the successor for r in R.

Algorithm 1

boolean, match ; member of R, r ;
$p \leftarrow$randomly chosen element of s ;
match $\leftarrow$**false** ; $r \leftarrow \alpha$;
while match = **false and** $r \neq w$ **do**
    if $a_p(r) = b_p(r)$ **and** $P(r) = T(r)$
    then match $\leftarrow$**true**
    else $r \leftarrow r'$;

Rabin and Karp remark that $P(r) = T(r)$ only needs to be tested where $a_p(r) = b_p(r)$. They also define a false match to have occurred wherever for an $r \in R$ we have $\Phi_p(P(r)) = \Phi_p(T(r))$ but $P(r) \neq T(r)$. Note that in their usage, a false match only indicates that an m-segment of the text needs to be compared against the pattern, character by character. Such false matches do not affect the output of the algorithm. When Algorithm 1 returns *match* := *true*, then a match has indeed been found, but these

false matches do impede the algorithm's speed. Rabin and Karp note that Algorithm 1 will be efficient for a class of string matching problems and for a given class of functions Φ_p, wherever the following three conditions hold:

1. The fingerprints of the strings can be represented much more compactly than the strings.

2. The probability of a false match occurring is small.

3. Fingerprints are easily updatable, i.e., given $a_p(r)$ and $b_p(r)$, computing $a_p(r')$ and $b_p(r')$ is easy.

Rabin and Karp comment that condition 2) ensures good expected running time for all instances of the class of string matching problems... "the randomization is not on the instances of the problem (such as pattern / text pairs), but rather in the choice of the fingerprint function. This choice is controlled by the algorithm so that only bad luck, not unfavorable instances of the problem, can produce a long running time."[3]

Subsequently we shall work out an example to illustrate this hashing approach to string matching, but first we elaborate somewhat on fingerprint functions, their form and efficient techniques for updating them.

A binary string $x = x_1 x_2 \ldots x_m$ may be regarded as the binary representation of the integer $H(x) = \sum_{i=1}^{m} x_i \cdot 2^{m-i}$. Rabin and Karp suggest that for

[3] "Efficient Randomized Pattern-Matching Algorithms" by Karp and Rabin, Technical Report TR-31-81-1, Harvard University, 1981.

any integer p, the function $H_p(x) = res(H(x), p)$ is a possible fingerprint function. We let $M = mt^2$ and define : $S = \{p | p \text{ is prime and } p \leq M\}$ with $\Phi_p(x) = H_p(x)$, for all x. In a corollary, the authors prove that when Algorithm 1 is executed with $S = \{p | p \leq mt^2 \text{ and } p \text{ prime}\}$, letting $\Phi_p = H_p$, then if we have that $mt \geq 29$, the probability of a false match occurring is $2.511/t$ for every instance of the input, i.e., $\{P(r), T(r), r \in R\}$. A sample calculation will illustrate how small this probability of a false match actually is. Assume a pattern length, m of 100 and a text of length $n = 10,099$. Here t has a value of $n - m + 1$ or just 10,000, and M equals $mt^2 = 100 \cdot (10,000)^2 = 10^{10}$. Since $p \leq M$, the range of Rabin and Karp's fingerprint function H_p is $\{0, 1, \ldots, p-1\}$ where $p \leq 10^{10} < 2^{34}$. Therefore each string of length 100 may be represented by a 24-bit fingerprint. The probability of a false match is $\leq .0002511$. When pattern and text are nearly equal in length, t which equals $n - m + 1$ will become small as compared to m. The authors recommend choosing $S = \{p | p \leq mn^2$, and p prime$\}$. If we do so the probability of a false match becomes at most $2.511/m$, *if* $mt \geq 29$.

The formula for updating the fingerprint for m-segments of the text is:

$$T(r+1) = (T(r) - 2^{m-1}T_r) \cdot 2 + T_{r+m}$$

Hence we have that :

$$a_p(r+1) = (a_p(r) + a_p(r) + \epsilon Y_r + Y_{r+m}) \bmod p$$

where $\epsilon = -2^m \bmod p$. Also whenever $0 \leq r_1, r_2 \leq p-1$, updating the mod function can be readily computed, i.e. $r_1 + r_2 \bmod p$ is either $r_1 + r_2$ or $r_1 + r_2 - p$.

Let us apply Algorithm 1 to a sample problem. We let the pattern $P = 1010110$ and as a text string T, we choose 0111001010110. Here pattern length m equals 7 and text length $n = 13$. The index set R, where occurrences of the pattern within the text string may occur is $R = \{1, 2, \ldots, 7\}$, indicating that t (the cardinality of the index set) is 7. In this example, we have that $mt = 7 \cdot 7$ which is 49. This is greater than 29 and hence the probability of false matches will be $\geq 2.511/7$ or less than .359). Once again we remind the reader, that this probability refers to instances wherein the algorithm must make a character by character comparison between the pattern and some m-segment within the text and *not* to occurrences of erroneous output being produced. As we noted earlier when t which is equal to $n - m + 1$ is small, which is the case when the pattern length is close in value to the text length, it is desirable to choose $S = \{p | p \leq mn^2$ and p prime $\}$. That is, our fingerprint function Φ_p will employ a modulus p which is to be a randomly chosen prime between 2 and mn^2 or $7 \cdot 13^2$ which is 1183. We will arbitrarily choose p equal to 29. Therefore, each fingerprint will be 5 bits long.

We begin by computing a fingerprint for the pattern $P = 1010110$. The function $H(P)$ treats the pattern as the binary equivalent of the integer

86. Next, compute $\Phi_p(P) = H_p(P) = res(86,29)$. When 86 is divided by 29, the remainder is 28, (more technically we would say that 86 is congruent to 28 modulo 29, written as $86 \cong 28(\text{mod}29)$) So that $\Phi_{29}(1010110)$ equals 28. Our required fingerprint is 11100, which is just the binary equivalent of 28. Next we compute a fingerprint for the first m-segment of the text, i.e., for the first 7 text characters which are $t_1t_2\ldots t_7 = 0111001$. We have $H_p(T(1)) = H_{29}(0111001) = res(57,29) = 28$. The fingerprint for $T(1)$ is 11100. $\Phi_p(P) = \Phi_p(T(1))$, a match is possible, therefore we compare $p_1p_2\ldots p_7$ with $t_1t_2\ldots t_7$. They do not match, hence no match is reported and the fingerprint for the second m-segment of the text, i.e., $T(2) = t_2t_3\ldots t_8$ is calculated. All calculations entailed in this example are shown in figure 5.1.

Referring to figure 5.1(h) we observe that the fingerprint for the pattern equals the fingerprint for $T(7) = t_7t_8\ldots t_{13}$. After a character by character comparison between $T(7)$ and the pattern, the algorithm would report that a match has been found at position 7 in the text string.

The reader should notice the relative ease of updating the fingerprint function as shown by the calculations in figure 5.1 for successive m-segments of the text.

We employed the update formula:

$$\Phi_p(T(r+1)) = (\Phi_p(T(r)) + \Phi_p T(r) + \epsilon \cdot t_r + t_{r+m}) \bmod p$$

Rabin and Karp mention that on a typical single-address computer, this

Pattern P = $p_1p_2p_3p_4p_5p_6p_7$ which equals 1010110; Text T = $t_1t_2t_3t_4t_5t_6t_7t_8t_9t_{10}t_{11}t_{12}t_{13}$ which equals 0111001010110; $\Phi_p(P) = H_p(P) = H_{29}(1010110) = res(86, 29) = 28$; therefore, fingerprint for pattern=11100

a) Calculation of fingerprint for pattern

$\Phi_p(T(1)) = H_p(T(1)) = H_{29}(t_1t_2 \ldots t_7) = H_{29}(0111001) = res(57, 29) = 28$; fingerprint for T(1)=11100

Observe that $\Phi_p(P) = \Phi(T(1)) = 11100$; i.e., the fingerprints agree. However, a character by character comparison between $p_1p_2 \ldots p_7$ and $t_1t_2 \ldots t_7$ is carried out. $p_1 \neq t_1$. Therefore, no match is reported.

b) Calculation of fingerprint for first m-segment of text, i.e., T(1).

$\Phi(T(r+1)) = (\Phi_p(T(r) + \Phi_p(T(r)) + \xi \cdot t_r + t_{r+m}) \bmod p$; $T(2) = (\Phi_p(T(1)) + \Phi_p(T(1)) + \xi \cdot t_1 + t_{1+7}) \bmod p = (28 + 28 + 17 \cdot 0 + 0) \bmod 29 = 56 \bmod 29 = 27$; fingerprint for T(2)=11011. This does not equal $\Phi_p(P)$, no match indicated.

c) Calculation of fingerprint for second m-segment of text.

$\Phi_p(T(3)) = (\Phi_p(T(2)) + \Phi_p(T(2)) + \xi \cdot t_2 + t_9) \bmod p = (27 + 27 + 17 \cdot 1 + 1) \bmod 29 = 72 \bmod 29 = 14$, therefore, fingerprint for $T(3) = 01110$, no match.

Figure 5.1: Algorithm 1 of Rabin and Karp. *Note that arithmetic is done in decimal here merely for ease of presentation.*

d) Calculation of fingerprint for T(3)

$\Phi_p(T(4)) = (\Phi_p(T(3)) + \Phi_p(T(3)) + \xi \cdot t_3 + t_{10}) \bmod p = (14 + 14 + 17 \cdot 1 + 0) \bmod 29 = 45 \bmod 29 = 16$, therefore, fingerprint for $T(4) = 10000$, no match.

e) Calculation of fingerprint for T(4)

$\Phi_p(T(5)) = (\Phi_p(T(4)) + \Phi_p(T(4)) + \xi \cdot t_4 + t_{11}) \bmod p = (16 + 16 + 17 \cdot 1 + 1) \bmod 29 = 50 \bmod 29 = 21$, therefore, fingerprint for $T(5) = 10101$, no match.

f) Calculation of fingerprint for T(5)

$\Phi_p(T(6)) = (\Phi_p(T(5)) + \Phi_p(T(5)) + \xi \cdot t_5 + t_{12}) \bmod p = (21 + 21 + 17 \cdot 0 + 1) \bmod 29 = 43 \bmod 29 = 14$, therefore, fingerprint for $T(6) = 01110$, no match.

g) Calculation of fingerprint for T(6)

$\Phi_p(T(7)) = (\Phi_p(T(6)) + \Phi_p(T(6)) + \xi \cdot t_6 + t_{13}) \bmod p = (14 + 14 + 17 \cdot 0 + 0) = 28 \bmod 29 = 28$, therefore, fingerprint for $T(75) = 11100$

Note that the fingerprint for $T(7)$ equals the fingerprint for the pattern. A character by character comparison between the pattern and $t_7 t_8 \ldots t_{13}$ verifies that a match has indeed been found. The algorithm returns match = true and the index 7.

h) fingerprint for T(7)

Figure 5.1: Calculation of fingerprints – continued.

updating operation can be performed with a constant number of steps. In fact, just 4 FETCHes, 3 ADDs, 3 COMPARISONs, 2 SUBTRACTIONs and 1 STORE are required. If we disregard the work entailed in choosing a random prime p from the set S, and the time required to verify matches, then Algorithm 1 solves the string-matching problem on-line in real-time. That is, they suggest that the pattern P and text T may be input bit by bit until P is exhausted, thereafter successive text characters are input. As the strings are being input, the algorithm would have to respond at each time instant, whether the last m text characters constitute an occurrence of the pattern. Their algorithm is on-line in the sense that the answer once j text characters have been submitted is ready before the $(j+1)^{st}$ text character is input. Moreover the response of Algorithm 1 for each successive text character input, requires no more than a constant amount of time, and is therefore operating in real-time (neglecting the effect of false matches, of course).

We have until now been considering the expected time behavior for Algorithm 1, and have commented that is $O(n)$, i.e., linear in terms of the text length. How does this algorithm perform in the worst case ? Consider a pattern $P = 1110100$ and let the text string $T = 0000000000000$. We let the prime p be 29 as in the prior example. Note that the pattern corresponds to a number which is a multiple of p, i.e., p corresponds to $(116)_{10}$ which is 4 times 29, our modulus. Hence $H_p(P) = H_{29}(1110100) = res(116, 29)$

which is 0, yielding 00000 as the fingerprint for this pattern. The fingerprint for $T(1)$, the first m-segment of the text, i.e., the string 0000000 will naturally also equal 00000. A false match will ensue. The calculations for each subsequent m-segment of the text will also yield fingerprints of 00000. That is, a false match will occur at *every* text position checked. Algorithm 1 will exhibit quadratic behavior in text and pattern length (actually, quadratic in text and fingerprint length). But this is not much better than the straightforward algorithm provided in chapter one !

To hedge against such a catastrophe, as the authors refer to this predicament, they provide Algorithm 2. Essentially Algorithm 2 chooses a new prime each time a false match is encountered . There will be an overhead attached to reinitialization, i.e., recomputing fingerprints requires $O(m)$ time and selecting a new random element of S takes $O(\log^2 M)$ time, (recall that $M = mn^2$). The probability that k or more false matches will occur when Algorithm 2 is employed is $\leq (2 \cdot 511/t)^k$. Its expected running time is still linear in text length.

Rabin and Karp provide a final version of their algorithm which is strictly real-time. When a match occurred in either of Algorithm 1 or Algorithm 2, a character by character comparison between the pattern and text was required to rule out the possibility of false matches. Their Algorithm 3 eliminates this portion of the algorithm, thereby allowing a small probability, which can be made negligible, of error. That is, Algorithm 3

may return the value match = true when in fact no match exists.

Their pseudocode is given below:

Algorithm 3

boolean, match ; member of R, r ;

$p_1, p_2, \ldots, p_k \leftarrow$independently chosen random elements of S ;

// here $p_i, i = 1, \ldots, k$ refer to primes and not to pattern characters //

match $\leftarrow$**false** ; $r = \alpha$;

while match = **false and** $r \neq w$ **do**

if $[a_{p_1}(r) = b_{p_1} \wedge \ldots \wedge a_{p_k}(r) = b_{p_k}]$

then match $\leftarrow$**true**

else $r = r'$;

Assume that Algorithm 1 has a probability ϵ for a false match, for some given problem. If Algorithm 3 returns the value match = false, we can be sure that no match exists. However, it may return the value match = true, when no match is present. The probability of this occurring is at most ϵ^k. If four primes are employed, and text length equals 10^6, then Rabin and Karp's real-time algorithm will return erroneous true matches with a probability of less than 10^{-24}. Obviously for much smaller text strings, this probability of error would no longer be insignificant. This might tend

to limit the applicability of their real-time algorithm. The authors do comment though that Algorithm 3 is especially fast when parallel processing is available. Fingerprints corresponding to different choices of p could of course be computed simultaneously . If the number of fingerprint functions employed were to be increased, then the probability of error could thereby be reduced, perhaps to tolerable levels even in the case of shorter text strings.

§5.2 *Improvements and Extensions*

Rabin and Karp suggest several improvements to their algorithms. One occurs in the choice of random primes from the set S (where S was chosen as the set of primes between 2 and M, where M equals mt^2 or mn^2). A modification might be to select a random integer in $[1, M]$ and then employ a probabilistic primality test algorithm (for example $[RABIN76]$). Another possibility is to settle for pseudoprimes in the choice of p. Every prime p satisfies $2^{p-1} \equiv 1 \bmod p$. However, it is possible for p to satisfy the above congruence and not be prime. We refer to such numbers as pseudoprimes to the base 2 (psp(2)). For large x, the number of psp(2)'s less than x is very small compared to the number of primes less than x. Rabin and Karp suggest modifying Algorithm 2 by taking

$$S = \{p | 1 \leq p \leq M \text{ and } 2^{p-1} \equiv 1 \bmod p\}$$

It is very likely that a p chosen in this manner will indeed be prime. And finally the authors maintain that provided M is large enough , the necessity of p being prime may be dispensed with altogether. These suggestions would certainly improve the performance of Algorithm 2.

The authors also provide a second family of fingerprint functions which employ 2×2 matrices instead of integers. A drawback entailed in the use of such matrices is that each fingerprint consists of four intergers mod p instead of just one. However, these matrices do admit to rather simple and eloquent updating techniques.

Rabin, in a second Technical Report [Rabin81] discusses a third choice for a family of fingerprint functions, he proposes the use of a randomly chosen irreducible (i.e., prime) polynomial $p(t) \in Z_2[t]$ of some small degree k (k prime makes the implementation more convenient). Then updating of fingerprints would entail performing mod pt arithimetic. Shift registers (with k-stages) and an exclusive-or operation over k-bit vectors would be sufficient hardware to implement the requisite calculations. Rabin comments that the operations are fast, the circuits entailed are simple, and little chip area would be required for VLSI implementations.

We have only discussed the applicability of the Rabin-Karp algorithms to the straightforward one-dimensional string matching problem. It is also possible to alter the index set R and the details of the fingerprint computations so that 2-dimensional pattern matching can also be efficiently

handled.

A restricted form of the two-dimensional string matching problem is the case wherein the pattern P is assumed is to be a square (i.e., an $s \times s$ array), as is the text T (an $n \times n$ array).

We are searching for all instances of the pattern within the text, i.e., we wish to find all (i,j) such that:

$$T(i-s+k, j-s+l) = P(k,l) \text{ for all } k,l \text{ such that } 1 \leq k \leq s,\ 1 \leq l \leq s$$

Refer to figure 5.2.

Rabin and Karp commence their search by positioning the $s \times s$ pattern in the highest left-most position of the text. The pattern is "marched down" until the last row is reached. The search recommences at the top but one position to the right (see figure 5.3).

We let $X(\langle k,l \rangle)$ be the string obtained by concatenating the rows of X together. That is, each of rows $X_1X_2 \ldots X_s$ are strings over $\{0,1\}^s$. We have that $X(\langle k,l \rangle) = X_1X_2 \ldots X_s$ is a string of length s^2. A fingerprint for this two-dimensional pattern $X(\langle k,l \rangle)$ may be readily calculated. Set $\rho = 2^s$, and then:

$$a_p(X\langle k,l \rangle) = H_p(X\langle k,l \rangle) = H(X_1)\rho^{s-1} + H(X_2)\rho^{s-2} + \cdots + H(X_s) \mod p$$

We let $Y(\langle k,l \rangle)$ be the first (i.e., upper-leftmost) $s \times s$ chunk of the text. Compute a fingerprint $b_p(Y\langle k,l \rangle)$ for this $s \times s$ segment in a similar manner. Rabin and Karp show how updating b_p for downward and rightward

marches may be readily accomplished. Their approach yields an $O(n^2)$ algorithm.

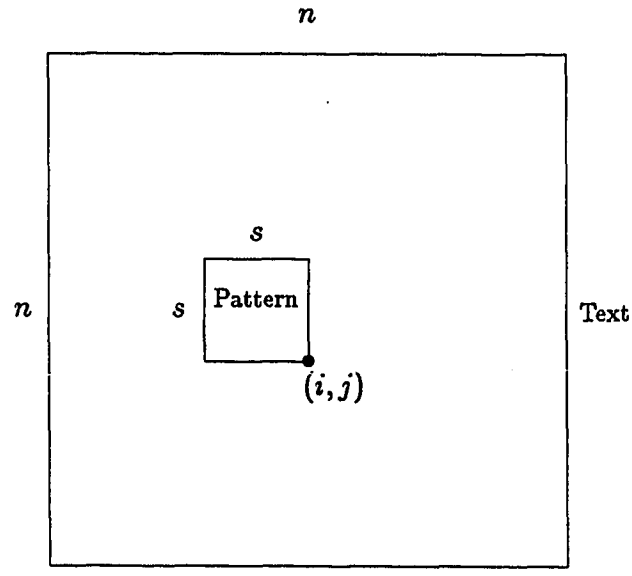


Figure 5.2: Two Dimensional Pattern Matching.

Baker [*BAKER78*] also developed an algorithm for two-dimensional pattern matching. His algorithm handles rectangular pattern and text strings but does not generalize well to irregular shapes. The algorithm of Rabin and Karp, on the other hand, readily generalizes to handle rectangular patterns, multi-dimensional patterns, and even irregular shapes.

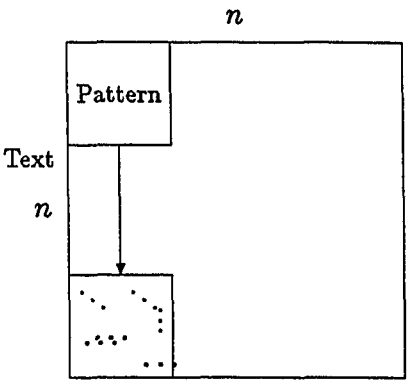


Figure 5.3(a) Initially the pattern is positioned in the highest left-most position of the text. Pattern is "marched down" until the last row is reached.

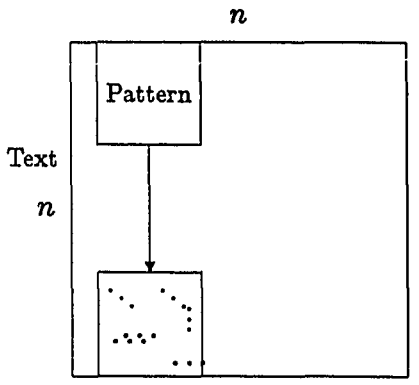


Figure 5.3(b) Search next begins in the top row, second column of text, and the pattern is once again marched down.

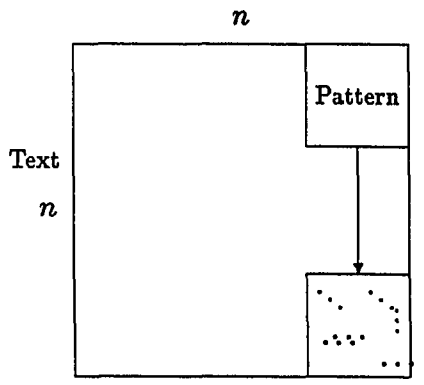


Figure 5.3(c) Finally the last s columns of the text are compared against the pattern, s rows at a time.

Figure 5.3: Pattern being marched across text.

CHAPTER 6

A PARALLEL ARCHITECTURE FOR STRING MATCHING

§6.1 *Introduction*

In the previous chapters of this work we have described various algorithms for the string matching problem. Our discussion began with a straightforward approach which required quadratic time in the worst case. The Knuth-Morris-Pratt and the Boyer-Moore algorithms, described in chapters one and two respectively, improved this time bound to $O(n)$. Essentially, the success of each strategy hinged upon a judicious pre-processing of the pattern which enabled optimal slides across the text string. Next we described two parallel approaches to the problem, each of which employed various models of a PRAM. Galil's algorithm, as we saw in chapter three, searched through the text for repeatedly larger prefixes of the pattern.

Vishkin's algorithm outlined in chapter four, used the concept of duels between various text positions to obtain a sparse set of suspicious indices, i.e., positions of the text wherein the pattern *might* begin. Finally in chapter five, we described the Rabin-Karp algorithm, unique in that the calculation of hashing functions (fingerprints) rather than character comparisons between the pattern and text was the basic operation. This last approach is noteworthy in that it is an essentially sequential algorithm, as are the KMP and Boyer-Moore approaches, however, unlike the latter two, the Rabin-Karp algorithm *is* readily parallelizable.

Our goal in this chapter will be to outline the architecture of a special-purpose parallel computer for string matching. We shall attempt to endow it with the requisite features to comfortably support Galil's parallel algorithm.

The first question that comes to mind is: "Why is a special-purpose parallel computer desired?". The obvious answer is speed of operation. The answer to the next and related question: "Why is a special-purpose parallel computer *required*, i.e., why won't a general purpose parallel computer suffice?" is not so obvious. R.Douglas in "*Algorithms + Alchemy = Architectures*" [*DOUG*85] addresses just this question, albeit in a more general setting. He comments that special-purpose serial computers are rarely required. Through what Douglas refers to as the alchemy of software, a general purpose computer is "molded" into a special-purpose processor

for an algorithm. First an algorithm is transformed into a program, "The program and the programming language define a virtual machine (VM) where the data structures, operations, and control constructs of the language represent memory structures, instruction and processor sequencing in the virtual machine."[1] This virtual machine program must be mapped onto an actual hardware machine (AM). Runtime support for the programming language, the compiler, and even the operating system, all assist in the VM to AM mapping. The AM itself may be implemented using an existing off-the-shelf general purpose processor implemented in microcode on an underlying body of hardware (e.g., gates and registers).

If the compiler does a poor job of mapping the VM onto the AM, or if the program does not match the AM well, (for an explanation of this so called *semantic gap* see [MYERS82]), then as much as a fourfold loss in thoughput may be observed. While this loss in execution speed is certainly not negligible, neither is it crippling. However, in the realm of parallel processing there is not yet a sufficient alchemy of parallel languages, compilers, and operating systems.

A bad match between a parallel algorithm and a parallel architecture can be devastating, effectively voiding any potential gain which was to have accrued from parallelism. It is this current lack of understanding of parallel processing which forces a software designer (one hesitates here to

[1] "Algorithms+Alchemy=Architectures" by Robert J. Douglas in Algorithmically Specialized Parallel Computers, edited by L. Snyder et al., Academic Press, Inc., 1985.

use software *engineer*) to become an architect.

Douglas observes that such specialized architectures are ad hoc solutions for particular algorithms, and as such they are expensive and time consuming. But as further justfication for their existence he describes "Amdahl's law", which in essence posits that any computation is composed of two parts. The first f_s, a serial component is executed in a slow sequential manner, and the second f_p, is that fraction of the algorithm which is amenable to fast parallel processing. As Douglas observes, Amdahl's law would seem to indicate there are two paths open to the researcher in parallel systems :

1) design systems wherein f_s is minimized and f_p maximized, and then endow the architecture with sufficient parallelism to expeditiously process f_p or,

2) construct a system which executes the f_s component very rapidly.

The visual system in mammals is cited as an instance of approach 1). A complex recognition task may entail an f_s of 10^2 sequential operations and f_p may be comprised of 10^7 or 10^8 parallel steps. The Cray 1 with a fast scalar processor is an example of a design which embodies approach 2). It is well known that the speed of light (about one foot/n sec) imposes a fundamental limit on the rate at which sequential computations may progress, and it is widely believed that we are fast approaching this limit. For this reason, Douglas believes that approach 1) holds greater promise.

It is not sufficient to choose an algorithm which minimizes f_s while it maximizes f_p, and then simply to utilize a machine with sufficient parallelism to effectively execute f_p. What is also critical is one must ensure that the algorithm to actual machine translation procedure introduces no new serial component f_s. Douglas provides an image processing example wherein improper appreciation of the above could cause the MPP (Massively Parallel Processor) [*BATCH*80] to stand idle most of the time because of inadequate forethought given to I/O considerations.

Special-purpose architectures are designed so as to minimize f_s for a particular algorithm or class of algorithms. At some point in the future it is to be hoped that an adequate software alchemy for parallel systems will exist so that an algorithm to actual machine transformation which minimizes f_s will be possible. Until then, special-purpose parallel computers will continue to flourish.

§ *6.2 Architectural Preliminaries*

We have been rather vague in our discussions of a parallel architecture "fitting" a particular algorithm. We will become more specific after a brief discourse on architecture. We are all familiar with sequential computers which are simply an embodiment of the Von Neumann architecture. Such a computer consists of two essential components, a Central Processing Unit

(CPU) wherein actual computations take place, and a memory unit where data and programs are stored (refer to figure 6.1). Note that the CPU itself is composed of two parts, an Arithmetic Logic Unit (ALU) which, as the name implies is where arithmetic and logical operations such as ANDing and ORing occur, and a control unit which is responsible for generating timing and control signals. Instructions are fetched one at a time (we are simplifying matters somewhat) from memory and brought into the CPU where they are decoded. Operands are brought in as required and the instructions then executed. Results are then written back into memory. At any computation step, a single instruction (say an ADD) is working on a single data stream (perhaps a pair of operands) for which reason, this model is ofter referred to as a Single Instruction/Single Data Stream (SISD) architecture [*HAYES*78].

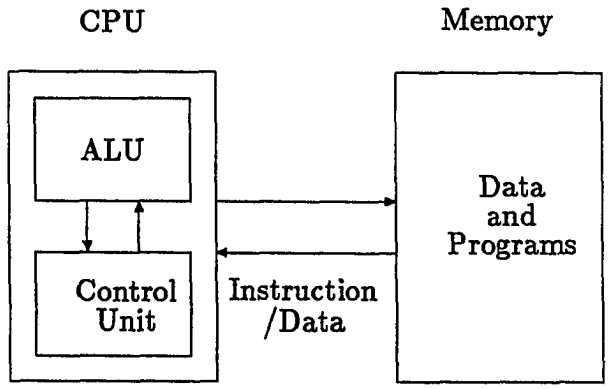


Figure 6.1: Von Neumann model of Computation.

One parallel model of computation is the Single Instruction/Multiple Data Stream (SIMD) computer (see [*HWANG*84]). An SIMD machine is depicted in figure 6.2. It consists of n Processing Elements (PEs), each PE consists of an Arithmetic Logic Unit (ALU) and a local Memory Unit. Comparing this situation with that depicted in figure 6.1, we observe that in an SIMD computer, there is no local control unit present within each PE. Instead, there is one control unit which decodes instructions for the entire melange. Instructions are then broadcast to the PEs, where in a lock step fashion each processor executes the specified instruction on its own local data, hence the name Single Instruction/Multiple Data Stream. It is possible to disable some processors from a particular computation step by the use of a masking scheme. If, for instance a mask bit within some PE is set to 1, that PE will participate in the current computation step, otherwise it would remain idle.

Frequently data residing within the local memory of one processor must be routed to that of another. It is the interconnection network which mediates such communication. To a large degree the speed of a parallel system hinges upon the forethought given to the design of its network. Ideally, every PE would have a direct link to every other PE, enabling any communication pattern to be accomplished in a single step.However, such

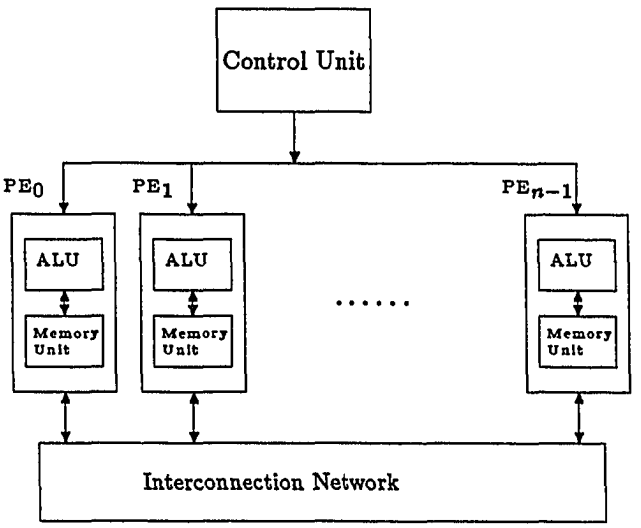


Figure 6.2: SIMD Machine Design.

a hardware instantiation of K_n, the complete graph on n vertices, ideal as it may seem from a performance standpoint, would require $O(n^2)$ bus lines. A typical SIMD machine may contain as many as 2^{14} (MPP – see [*BATCH*80]) or even 2^{16} (Connection Machine – see [*HILL*85]) PEs. Even with the plummeting of hardware costs, the price tag for such an interconnection network would indeed be astronomical. At the other end of the cost/performance spectrum one would find the UNIBUS structure. A single bus line connects all components, thereby enabling communication between any pair of devices. This format was successful in the Digital PDP series of computers [*BELL*71]. The complexity of serial computations is

usually predominated by the actual processing time required, whereas the performance of parallel algorithms is often determined by both the quantity and format of communications entailed. One can readily envision the bottleneck of monstrous proportions that would likely ensue from an SIMD computer with 2^{16} processors endowed with but a single bus line. The interconnection networks for actual SIMD systems usually lie somewhere between the two extremes just cited. Each processor in the system will be connected directly to some subset of the remaining PEs. Arbitrary interprocessor communication patterns generally will require multiple passes through the network, with data items passing through some number of intermediate addresses before reaching their proper destinations. There is literally a cornucopia of interconnection networks to choose from. Three of these, the perfect shuffle, cube, and mesh interconnection schemes will be described shortly. A nice introduction to interconnection networks may be found in [*SIEG*79]. But first we give a more formal definition of an SIMD machine. We adopt the schema presented in a recent book by Siegel on the subject [SIEG85].

An SIMD machine may be formally defined as a five-tuple, (N,C,I,M,F), where:

1. N is positive integer that represents the number of processing elements in the computer. As mentioned above, typical values for N on present day systems may range from 2^{14} to 2^{16}.

2. C is the set of control unit instructions. These include the instructions which the control unit executes such as *for, until, step, do,* that control program loops.

3. I is the set of PE instructions, i.e., those instructions that each processing element which is activated will execute. Instructions to transfer data between various PE registers would be included in I.

4. M is the set of masking schemes. The purpose of a mask as cited earlier is to activate only those PEs which are to be included in the execution of a particular instruction. As Siegel notes, M includes PE address and data conditional masks.

5. F is the set of interconnection functions that comprise an interconnection network. Each member of F is a bijection on the set of processor addresses, i.e., $\{0, 1, \ldots, n-1\}$. F determines the communication links between PEs. Several examples will be described shortly.

This five tuple notation (N,C,I,M,F) provides us with a means of specifing a particular SIMD architecture.

We commented previously that the interconnection network for an SIMD machine will have a large bearing on its performance. Three which have received widespread attention in the literature are the mesh, perfect shuffle and cube networks. The mesh network is perhaps the easiest of these to visualize. The N processors which comprise the system are arranged as

a square with $\sqrt{N}$ PEs on a side (see figure 6.3).

Each PE (except those on the border) has four neighbors, e.g., north, south, east, west, with which it may communicate directly. The lines leading out of the PEs in the same direction may be viewed as a function defined over the address set of the machine, e.g., $\{0, 1, \ldots, 15\}$ in our example.

We denote the four functions in this network as N, S, E, and W where:

N(i)= $i - \sqrt{N}$ or just $i - 4$ in this example, where $4 \leq i \leq 15$. For instance $N(10) = 6$ and $N(15) = 11$.

S(i)= $i + \sqrt{N}$ which corresponds to $i + 4$ here, with i ranging from 0 to 11. We have $S(8) = 12$ and $S(3) = 7$.

E(i)= $i + 1$. $E(8) = 9$ and $E(6) = 7$. Here, i is restricted to addresses of PEs in the leftmost three columns, i.e., E(i) is undefined for $i = 3, 7, 11$, or 15.

W(i)= $i - 1$. $W(14) = 13$ and $W(2) = 1$. W is defined only for the rightmost three columns.

When a route instruction is executed on an SIMD machine, all PEs would send data in the same direction, i.e., execute the same interconnection function. We observed in the definitions for N, S, E, and W certain restrictions in their applicability. It is sometimes convenient to embellish the mesh network with "end around" connections as well. For instance PE_0, PE_1, PE_2, and PE_3 would be connected to PE_{12}, PE_{13}, PE_{14}, and PE_{15} respectively

by the N function. The applicability of the functions S, E, and W would be similarly extended by such additional bus lines. The mesh network endowed with end around connections was the basis for the Illiac [*BARN*68] and MPP [*BATCH*80] computers.

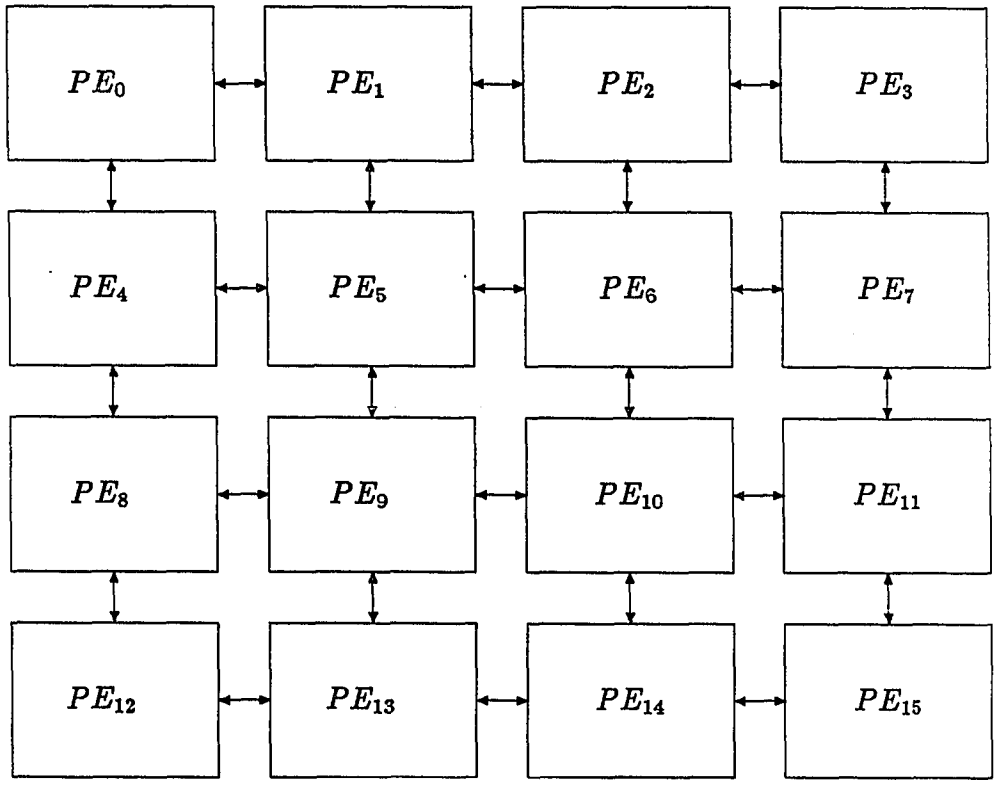


Figure 6.3: A mesh interconnection network.

The second interconnection network to be described is the perfect shuffle (see figure 6.4). Here the set F of interconnection functions consists of PS, the perfect shuffle interconnection and EX, the exchange function. In figure 6.4, PS connections are denoted by lines with single arrows whereas the EX connection lines contain double arrows. These two functions are defined below.

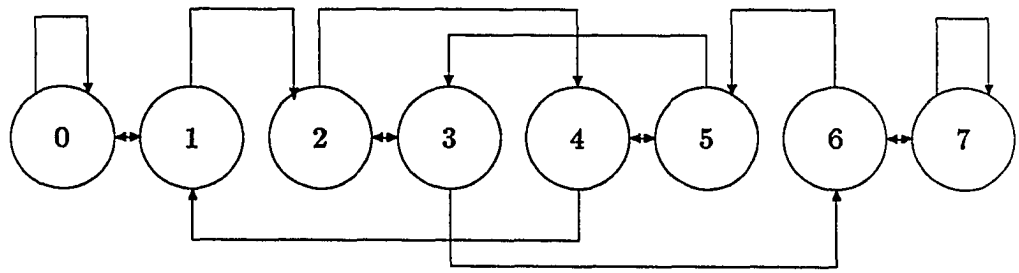


Figure 6.4: Perfect Shuffle Interconnection network.

$PS(i) = 2i \bmod (N-1)$, for example $PS(2) = 2 \cdot 2 \bmod 7$ or 4, and $PS(5) = 2 \cdot 5 \bmod 7$ or 3. Note though that PS(7) equals 7.

$$EX(i) = \begin{cases} i+1 & \text{if } i \text{ is even} \quad \textit{So that } EX(0) = 0+1 \textit{ or } 1. \\ i-1 & \text{if } i \text{ is odd} \quad \textit{Whereas } EX(5) \textit{ equals } 5-1 \textit{ or } 4. \end{cases}$$

It is sometimes convenient to include a third function, the inverse shuffle, denoted PS^{-1} as a member of the interconnection network for a perfect shuffle computer (PSC). As the name implies the inverse shuffle is

the inverse of the PS function, i.e., $PS^{-1}(j)$ equals that value i such that $PS(i) = j$. Pictorially, the inverse shuffle would correspond to the shuffle lines in figure 6.4, but with the direction of the arrows reversed. For example, $PS^{-1}(3) = 5$ and $PS^{-1}(6)$ would equal 3. Let $a_{m-1}a_{m-2}\ldots a_1a_0$ represent the binary address of processor i. The effect of a perfect shuffle is to map address i into address $PS(i) = a_{m-2}a_{m-3}\ldots a_1a_0a_{m-1}$, this corresponds to a circular left shift of the bits composing the address. Similarly, the effect of an inverse shuffle corresponds to a circular right shift, i.e., when $j = a_{m-1}a_{m-2}\ldots a_1a_0$, then $PS^{-1}(j)$ would equal $a_0a_{m-1}a_{m-2}\ldots a_2a_1$.

The term perfect shuffle derives from card playing lore. A perfect shuffle of a deck would correspond to alternately choosing cards from each half of the deck (see figure 6.5). The perfect shuffle forms the basis for the NYU Ultracomputer [$SCHWA80$].

Once more, let $a_{m-1}a_{m-2}\ldots a_i\ldots a_1a_0$ denote the binary address for some processor j. The $CUBE_i(j)$ function connects PE_j to processor $a_{m-1}a_{m-2}\ldots\overline{a_i}\ldots a_1a_0$, where $\overline{a_i}$ is the complement of a_i. In general a cube network will consist of m such functions, one for each bit position, where $m = \log_2 N$ and N is the number of PEs in the system. A cube interconnection network with $N = 8$ PEs is shown in figure 6.6. For clarity the set of 8 processors have been drawn 3 times to illustrate each of the 3 cube functions. Referring to figure 6.6 we observe that $CUBE_0(4) = 10\bar{0} = 101$ or 5. Also, $CUBE_1(3) = 0\bar{1}1 = 001$ which is 1 and $CUBE_2(7) = \bar{1}11$ which

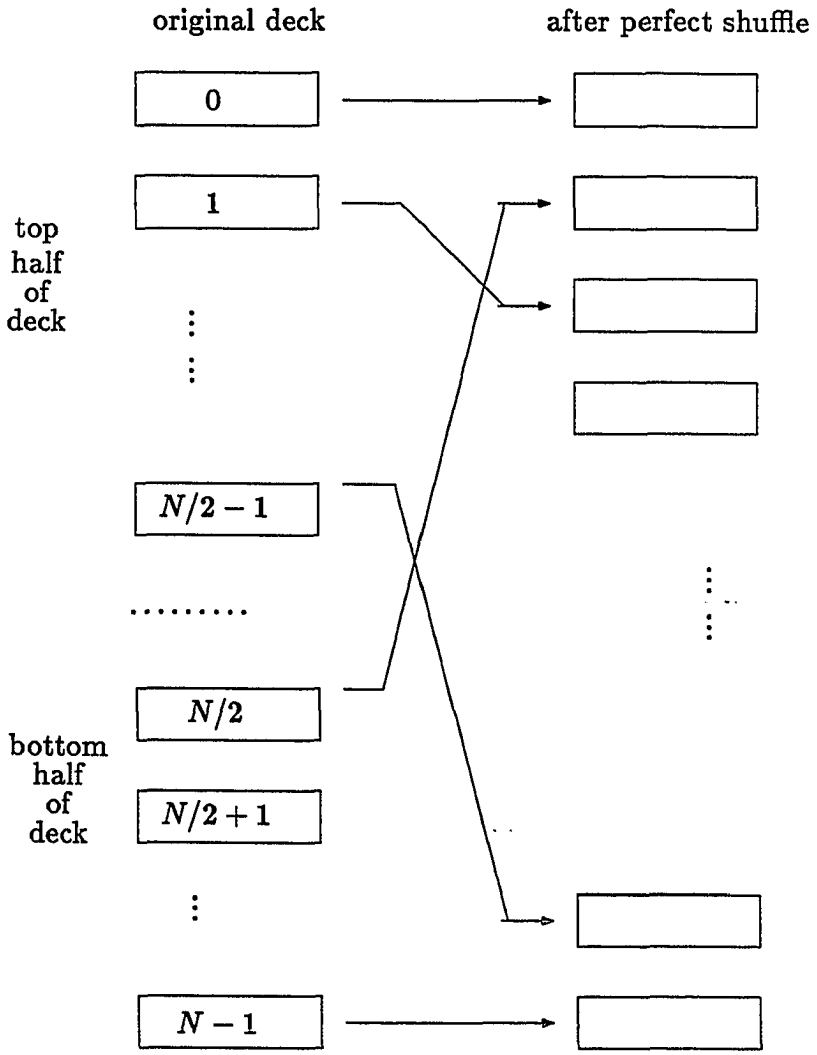
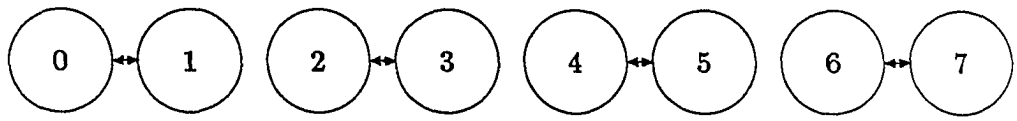


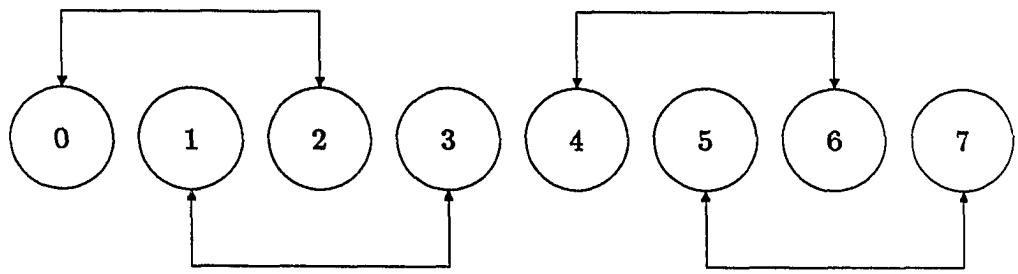
Figure 6.5: Perfect shuffle of a deck of cards.

is 011 or just 3. As with other interconnection networks, once a specific function f_i, from the set F has been selected then all PEs which are enabled

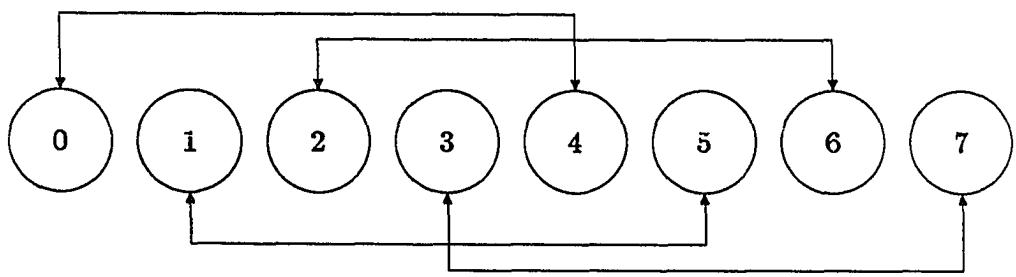
will route data using the same f_i. So that on a cube network only one of the $CUBE_i$ functions may be used at a time.



a) $CUBE_0$ connections.



b) $CUBE_1$ connections.



c) $CUBE_2$ connections.

Figure 6.6: Cube interconnection network.

The name for this interconnection scheme stems from the fact that PEs may be envisioned as comprising an m-dimensional cube (see figure 6.7). The $CUBE_0$ function corresponds to horizontal edges, the $CUBE_1$ interconnection to diagonal lines, and the $CUBE_2$ mapping to those PEs connected in a vertical fashion. In the mesh and perfect shuffle interconnections, the number of connecting lines required per processor remains fixed regardless of system size N. That is F, for a mesh connected computer (MSC) contains 4 functions (e.g. N, S, E, W) whether the SIMD machine contains 2^6 PEs as in the case of the Illiac or 2^{14} PEs as in the MPP. Similarly, a PSC would have 2 connections per PE (or 3 if the inverse shuffle is included) no matter what its size. This "fixed degree" property of the mesh and shuffle interconnections eases VLSI inplementations and makes expandability of SIMD computers employing them, less cumbersome. As is evident from figure 6.7, the number of CUBE functions in F would grow as the logarithm of N, making the cube interconnection somewhat costly in large scale systems. Preparata and Vuilleman [PREP80] have discovered a network, the Cube Connected Cycles, which combines the flexibility of the Cube network with the fixed degree property of the mesh and shuffle schemes. The Cube Connected Cycles also requires less VLSI layout area than the best schemes for the shuffle exchange network [*LEIGH*83].

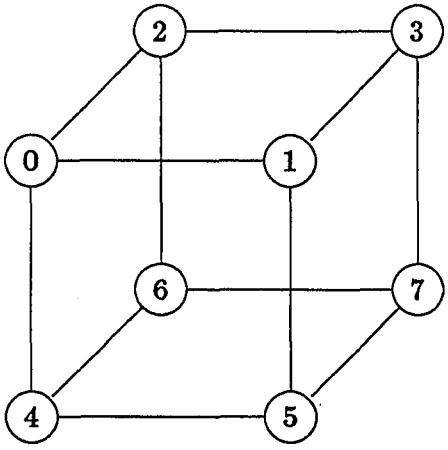


Figure 6.7: Cube interconnection network - alternate representation.

§ *6.3 Architecture – algorithm fit: an example*

Multiplication of matrices is a problem which arises in many arenas of Computer Science. It is at the crux of transitive closure algorithms in graph theory [$AH074$], plays a fundamental role in Valiant's parsing algorithm [$HARR78$], and occurs naturally in solving Linear Programming problems [$HILL74$]. Until the late 1960's it was believed that $O(n^3)$ time was a strict lower bound for multiplying $n \times n$ matrices. However, in 1969, Strassen discovered a divide and conquer strategy which effectively lowered this bound to $O(n^{2.81})$. Subsequent work by Pan lowered the requisite time even further to $O(n^{2.795})$ (see [$AHO74$]). A description of Pan's work which

employs trilinear forms, may be found in a recent book on sequential and parallel algorithms by Lydia Kronsjö [*KRON*85].

We let A and B represent two $n \times n$ matrices. Then their product $C = A \times B$ is given by:

$$C_{i,j} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad \text{for} \quad i,j = 1,\ldots,n$$

It is apparent that three nested loops are contained in the computation of C. The innermost loop contains the expression:

$$(*) \qquad C_{ij} = C_{ij} + a_{ik} b_{kj}$$

Kronsjö [*KRON*85] observes that by varying the loop indices i, j, k it is possible to obtain six different algorithms. As she comments, it is not that operations are being performed in a different order, for indeed they are not, but rather that memory access patterns will differ in each case, and it is this latter factor which affects the performance of an algorithm on a particular parallel machine.

We will present three matrix multiplication algorithms in abbreviated form to illustrate the "algorithm - architecture fit" that we've been alluding to in this chapter. In the first, the loop is in the form i,j,k:

Algorithm Matrix ProdOne (A,B: Matrices)

cobegin i : $\langle 1 : n \rangle$
  cobegin j : $\langle 1 : n \rangle$
    cobegin k : $\langle 1 : n \rangle$
      temp(i, j, k) $\leftarrow a_{ik} \times b_{kj}$
    coend
    $c_{ij} \leftarrow$ *innersum* (temp(i, j, 1:n))
  coend
coend

Kronsjö explains that *innersum* is a function whose argument, temp(i, j, 1:n) is a vector, and whose result is the sum of the components of this vector. A partial sum method which adds components from the two halves of the vector together is employed, producing a vector with half the length of its argument. This is done recursively until a single element remains. This matrix multiplication algorithm has a theoretical time complexity of $O(\log n)$ and requires $O(n^3)$ space. For an efficient implementation, a three-dimensional network would be required. Clearly, an MCC with its two-dimensional mesh interconnection scheme is not equal to the task, hence in this case we would have a poor algorithm - architecture fit.

We describe a second algorithm for which our MCC also would not be

ideally suited. It is possible to transform forms ikj and jki into a parallel algorithm which employs a broadcasting (i.e., fast copying) mechanism of the parallel machine. Let A and B be the two 2×2 matrices given below:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad , \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

One vector from each of the two matrices is selected and these are duplicated n (e.g., 2) times to form new matrices:

$$\begin{pmatrix} a_{11} & a_{11} \\ a_{21} & a_{21} \end{pmatrix} \qquad \begin{pmatrix} b_{11} & b_{12} \\ b_{11} & b_{12} \end{pmatrix}$$

Note that we have chosen to broadcast the first column of A and the first row of B. If we multiply the corresponding entries from these two matries, we obtain:

$$(**) \qquad \begin{pmatrix} a_{11}b_{11} & a_{11}b_{12} \\ a_{21}b_{11} & a_{21}b_{12} \end{pmatrix}$$

In general, as Kronsjö points out, the vectors are chosen so that the ij^{th} element of the matrix forms one of the terms of the inner product matrix. Therefore, next we broadcast the second column of A and the second row from B:

$$\begin{pmatrix} a_{12} & a_{12} \\ a_{22} & a_{22} \end{pmatrix} \qquad \begin{pmatrix} b_{21} & b_{22} \\ b_{21} & b_{22} \end{pmatrix}$$

Their product yields:

$$(***) \qquad \begin{pmatrix} a_{12}b_{21} & a_{12}b_{22} \\ a_{22}b_{21} & a_{22}b_{22} \end{pmatrix}$$

Adding $(**)$ to $(***)$, we obtain the final product matrix:

$$C = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}$$

The pseudocode for this method is contained in [$KRON85$]. The algorithm requires $O(n)$ time and $O(n^2)$ space. As we indicated above this approach will not yield an efficient implementation unless the available parallel computer is endowed with a broadcast facility, effectively ruling out a MCC as the machine of choice. However, this algorithm would seem to be ideally suited to the ICL (International Computers Limited) DAP (Distributed Array Processor). The interconnection network F in the DAP consists of the four functions which constitute the wrap-around version of the mesh network, i.e., the Illiac network. And in addition F includes a rapid copy feature, which is based upon registers which are globally accessible. The DAP is a bit processor composed of 2^{12} PEs arranged in a square. It is possible to disable part of the 64×64 melange of processors

via masking. Several articles describing the DAP and matrix algorithms which run well on its architecture may be found [$PAD84$].

Finally, we mention Cannon's algorithm. As Kronsjö reports it also requires $O(n)$ time and $O(n^2)$ space. It is somewhat similar to the last method. However, instead of employing a broadcast mechanism, Cannon's algorithm cyclically shifts rows from A and columns from B. The rows of A are shifted west while B's columns are shifted north in order to obtain proper alignment between those terms which constitute part of the inner product for each element of the product matrix. It would appear that our MCC does indeed possess the appropriate architectural features to effectively support an implementation of Cannon's algorithm.

§ *6.4 Stages of an algorithm*

D.Gajski, addressing himself to the general-purpose vs. special-purpose computer issue, discusses the natural evolution which he believes governs problems and their solutions...

> "Firstly, the model for some natural phenomenon is defined and the problem specified. Then, for a while many people work on it until a few optimal solutions emerge. The problem has matured, when we know all about it and we can make tradeoffs between different methods of solving it...

> Special-purpose computers are designed to solve mature problems"[2]

To substantiate Gajski's claims, we consider the problem of sorting. In the 1950's and 1960's when Computer Science was still in its infancy, sorting algorithms had already become the subject of widespread study. At first, optimal sequential algorithms were discovered, e.g. Heapsort and Mergesort. For a thorough treatment of sequential sorting algorithms, the reader is referred to volume 3 (*Sorting And Searching*) of Knuth's "encyclopedia" of Computer Science [$KNUTH73$].

The next milestone occurred in 1968. Batcher proposed an algorithm which was amenable to parallelism, in fact, he also proposed the first sorting network [$BATCH68$]. His parallel algorithm gainfully employed properties of bitonic sequences. Roughly, a sequence of numbers is bitonic if the numbers first increase and then decrease. For example, the sequence 17, 24, 35, 9 is bitonic. The building blocks of Batcher's sorting network are comparison elements which are capable of comparing two numbers and outputting them in sorted or reverse order. Unfortunately, his network, which may be viewed as a special purpose processor for sorting, required $O(n \log^2 n)$ of these elements, effectively making its implementation infeasible in light of hardware costs in the late 1960's.

In 1971 Stone proposed a sorting processor for Batcher's algorithm

[2] "Does General Purpose Mean Good For Nothing?" by Daniel D. Gajski, in Specialized Parallel Computers, edited by L.Synder et. al.; Academic Press, Inc. 1985 p. 249.

[$STONE71$]. A perfect shuffle interconnection network was employed by Stone to lower the requisite hardware for a sorting network to $O(n)$. His design entailed the use of $n/2$ Compare-Exchange modules which could alter their mode of operation under the control of a recirculated mask. Hence, a single level of comparators could simulate the entire Batcher network, one level at a time. Stone's approach required $O(\log^2 n)$ time amounting to a *cost* of $O(n \log^2 n)$.

Since 1971, special-purpose parallel sorting processors employing mesh-connected computers, cube connected computers and tree machines have been developed. Algorithms for shared-memory SIMD computers and even for MIMD machines have also been discovered. An excellent treatment of parallel sorting algorithms may be found in a recent book by Akl [$AKL85$]. Clearly, under Gajski's criteria, sorting would be deemed a mature problem.

Our claim here, is that the problem of string matching is itself reaching the stage of early maturity. Optimal sequential algorithms were discovered in the 1970's, e.g. the KMP algorithm in 1974 and the Boyer Moore approach in 1977. Various extensions of the basic problem to two-dimensional patterns [$BAKER78$], multiple string searches [$AHO75$] and even approximate string matching [$HALL80$] have been explored. A systolic based string matcher was developed in the late 1970's by Foster and Kung [$FOST80$]. A more recent special purpose processor based upon serial techniques was built by Proximity, Inc. [$YIAN83$]. This latter ma-

chine was capable of obtaining approximate string matches as well. Parallel approaches to the basic problem have also been discovered, e.g. Galil's algorithm in 1983 and Vishkin's in 1984. We believe that the problem of string matching presently, is at the same juncture at which sorting found itself between 1968 and 1971. Furthermore, we believe that the time is ripe to explore the feasiblity of a special purpose parallel processor for string matching.

§6.5 *A special – purpose parallel processor for string matching*

In this section we describe features which we believe a special-purpose parallel computer for Galil's algorithm should possess. We have decided upon this particular algorithm for several reasons. One being simply that his was the first parallel string matching algorithm. More importantly, though is the clearcut manner in which the performance of Galil's algorithm hinges upon an efficient data broadcasting mechanism — an attribute that we believe to be at the heart of any comparison based string matching approach. Characters must be dynamically routed though the system, so that pattern/text comparisons may be made expeditiously. As further justification for our choice, we cite a remark by Dechter and Kleinrock stating that algorithms which employ broadcasting have not received sufficient attention in the literature [$DECH86$].

Galil's algorithm is a synchronous parallel algorithm which suits an SIMD environment well. The "parallel string processor" (PSP) that we are proposing will consist of $n = 2^{16}$ PEs. Because of its fixed interconnection degree (three bus lines per PE) we have decided on a perfect shuffle interconnection. However, a Cube Connected Cycles interconnection would have also been feasible. A block diagram of our string matching machine is given in figure 6.8. Each PE has a local memory of 4096 bits as well as registers which will be described shortly. We assume that a pattern or text character may be readily encoded using eight bits (e.g. in EBCDIC). Each local memory may therefore hold up to 512 string characters. Consequently our machine has the capability of handling pattern/text problems on the order of 2^{25} or about 32 million "bytes". For larger data base problems, the local memories may be suitably augmented.

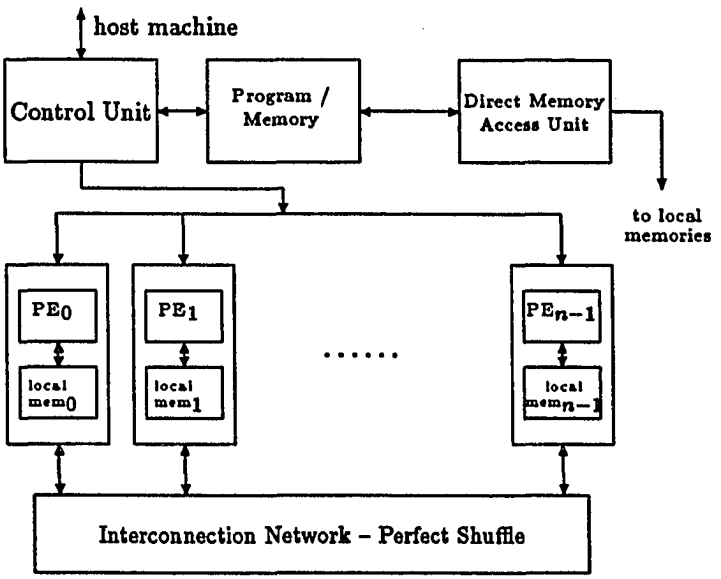


Figure 6.8: A block diagram of the PSP.

We envision two possible environments for the PSP. In the first, our machine serves as a back-end processer to a host computer. Its sole function, being to solve string matching problems. The second possibility arises if one already possesses a general purpose parallel computer with 2^{16} or more PEs (e.g. a Connection Machine), then our PSP may be adapted to run as a "utility package" (analagous to a sort routine on present day systems). For ease in the subsequent presentation, we assume the first scenario. Naturally, if the second situation holds, then the processing elements would be somewhat more sophisticated than the PEs we describe below.

A typical processing cell is shown in figure 6.9. All of the registers used are detailed in table 6.1.

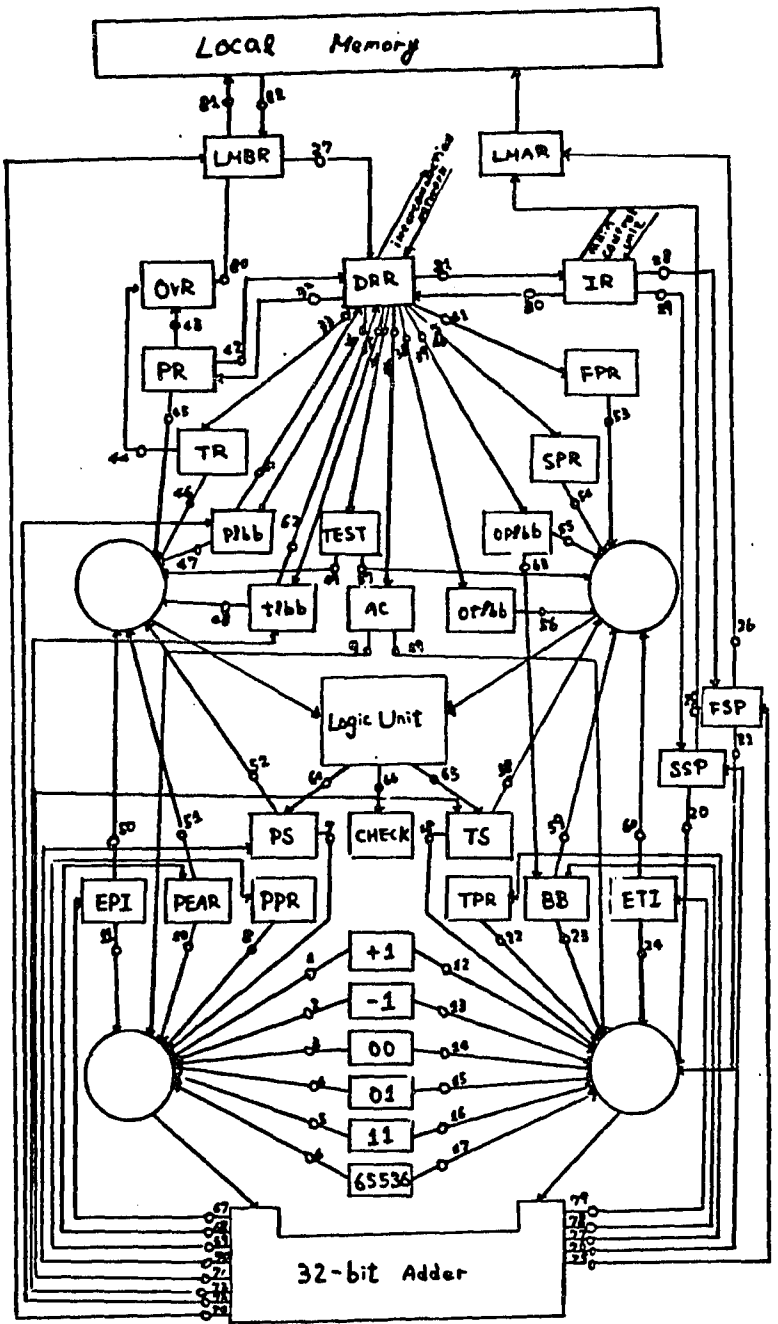


Figure 6.9: Register Set for a PE in the PSP Data Paths are also drawn.

Table 6.1: Registers in the Parallel String Processor(PSP).

DRR- Data Route Register — used for broadcasting messages between PEs. (8 bits)

PR- Pattern Register — the first m PR registers will be loaded with the entire pattern, one character per PE. (8 bits)

TR- Text Register — holds one text character. The entire text is stored in $PE_0 \cdots PE_{testlen-1}$. (8 bits)

FPR- First Pattern Register — holds a pattern character to be tested against contents of PR or TR. (8 bits)

SPR- Second Pattern Register — holds an additional pattern character for subsequent testing against PR and TR contents. (8bits)

PEAR- Processing Element Address Register — holds a PE's address within the melange. (16 bits)

PS- Pattern Switch — indicates potential matches between positions in the pattern with prefixes of this string. (1 bit)

TS- Text Switch — indicates positions of text which match prefixes of pattern. (1 bit)

PLBB- Pattern Local Bulletin Board — indicates positions within a block where matches in the pattern string have been found. (16 bits)

TLBB- Text Local Bulletin Board — indicates positions within a block where matches in the text string have been found. (16 bits)

OPLBB- Other Pattern Local Bulletin Board — used to communicate with neighboring PLBBs. (16 bits)

OTLBB- Other Text Local Bulletin Board — used to communicate with neighboring TLBBs. (16 bits)

EPI- End of Pattern Indicator — will indicate PE containing last pattern character. (2 bits)

ETI- End of Text Indicator — will indicate PE containing last text character. (2 bits)

LMBR- Local Memory Buffer Register used on READs and WRITEs. (16 bits)

FSP- First Stack Pointer — when pattern exceeds 2^{16} characters the FSP will be used to fetch subsequent segments of pattern as required. (12 bits)

SSP- Second Stack Pointer — when text exceeds 2^{16} characters the SSP will be used to fetch subsequent segments of the text as necessary. (12 bits)

BB- Bulletin Board — if the algorithm is in the periodic mode, this register

will indicate period size P, and the length of the periodic string, L. (48 bits)

LMAR- Local Memory Address Register — loaded from either the FSP or SSP, will specify local memory addresses for READs and WRITEs. (12 bits)

PPR- Pattern Pointer Register — used to indicate indices within the pattern string, useful when pattern length exceeds 2^{16}. (28 bits) - Actually since pattern length should never exceed text length 27 bits would have sufficed.

TPR- Text Pointer Register — used to indicate indices within text string, useful when text length exceeds 2^{16}. (28 bits)

IR- Instruction Register — receives decoded instructions from the main control unit. (83 bits if a strictly horizontal microprogramming approach is taken)

OVR- Overflow Register — used to store characters in local memory when pattern and text lengths exceed 2^{16}. (8 bits or 9 bits – see subsequent explanation)

TEST- Test Register — used to hold constants required for masking. (32 bits) May hold two 16 bit numbers.

AC- Address Change Register — used to modify PEAR when a circular shift of addresses is required. (16 bits)

CHECK- Check Register — holds the results of a test performed by the Logic Unit. (1 bit)

In addition there are six read only registers that contain constants. These registers are not to be used as destinations, but merely as source registers. The six constants are +1, -1, 00, 01, 11, and 65,536.

As specified in table 6.1, the Text Register (TR) will hold a single text character with TR_i containing the i^{th} symbol from the text string. When text length exceeds N, which is 2^{16} in our proposed machine, t_j will reside in the k^{th} word from the top in the local memory of $PE_j \bmod n$ where $k = \lfloor j/n \rfloor$. The SSP (Second Stack Pointer) will supply addresses to the LMAR (local memory address register) for necessary READs and WRITEs.

Referring once again to figure 6.8, we observe a Direct Memory Access Controller (DMAC). In the case wherein text length exceeds 2^{16}, we desire that the processors begin their work *while* the local memories are being loaded. The DMAC permits this pipelining (overlap) of functions.

Initially we assume that the pattern $P = p_1 p_2 \ldots p_n$ is present in the pattern registers (PRs) of the first n PEs, one pattern character per processor. The case wherein pattern length exceeds 2^{16} is handled in a manner

similar to the "text overflow" case above. However, in the case of "pattern overflow", the LMAR would be loaded from the FSP (First Stack Pointer). The ensuing situation in each local memory is similar to the traditional algorithm for maintaining two stacks in one array, see [KNUTH68].

As cited above, each local memory will contain an address register (LMAR), and a buffer register (LMBR) to mediate communication with local memory units. Each PE is assumed to know its own address. This information may be loaded from the control unit at startup time, or the PEs themselves may be enlisted in the requisite calculations. Also we assume that the PE addresses may be changed under program control. The Data Route Register (DRR) or the PE Adder could be used toward this end.

Since we permit pattern and text lengths in excess of 2^{16}, two additional registers, a Pattern Pointer Register (PPR) and a Text Pointer Register (TPR) are required, to specify indices within the pattern and text respectively. Each time a new character is fetched from the local memories of the PEs, the contents of the PPR or TPR will be incremented by 2^{16} to maintain correct indexing within strings. However, for the most part, to help simplify the subsequent discussion, we shall ignore the details entailed in handling such large strings. For instance, values of SWITCH registers would have to be stored in memory necessitating a $9-$ bit instead of $8-$ bit word size. We believe that a Technical Report would be the correct place to cover such specifics.

There are numerous decisions that arise as one sets out to design a computer. Simulation studies often guide an architect toward correct choices. For example, the reader may recall (from chapter three) that the input to Galil's algorithm is a string Z of the from $X\$Y$. The length of X, which is the pattern is n. The length of Y, the text string is assumed to be twice the pattern length and the \$ is used as a delimiter. Hence the length of Z is 3n+1. One version of the algorithm runs on a PRAM (a definition of which may be found on page 33 of this work) with 3n+1 processors. Each processor of the PRAM is responsible for one character of Z and one entry of an array SWITCH, also of length 3n+1. When the algorithm is done, $SWITCH(i)$ will equal 1 iff an occurrence of the pattern X has been found at $Z(i)$, i.e., position i within the Z array. We have decided instead, to place the pattern in the pattern registers (PRs) of $PE_0PE_1\cdots PE_{n-1}$ and the text is to reside in the text registers (TRs) of $PE_0PE_1\ldots PE_{2n-1}$. This should help to diminish local memory accesses for large strings. However, this decision necessitates the use of two $SWITCH$ arrays within each processor. The pattern $SWITCH(PS)$ will refer to that portion of Z within the pattern itself, and the text $SWITCH(TS)$ to the text component of Z. The relationship between Galil's array and ours is provided below:

$$SWITCH(1)\cdots SWITCH(3n{+}1) = PS(0)\cdots PS(n{-}1)0TS(0)\cdots TS(2n{-}1)$$

Observe the 0 in the $(n+1)^{st}$ position of our array, immediately after $PS(n-1)$, this corresponds to the location in Galil's scheme which contains

the \$; certainly no occurrence of the pattern can commence here.

A second decision in the design of our PSP was to have each PE hold a single text character. Galil had described optimal cost versions for his algorithm wherein each processor would be responsible for $\log n$, or even $\log^2 n$ symbols. In our machine this would necessitate on the order of an additional 16 or 256 registers per PE or more frequent local memory accesses (though naturally fewer PEs would be required). It should be evident that an optimal design strategy for our SIMD machine must await simulation studies.

Referring once again to figure 6.9, there are several other registers that merit discussion. Two such registers are employed for delimiter purposes, each consists of two bits. The end of pattern indicator (EPI) and end of text indicator (ETI) will signal the end of those respective strings. When EPI=00, for some processor j, this will specify that PE_j contains the last pattern character. Whereas an EPI value of 11 indicates that the pattern continues into the next processor. When EPI contains 01, a fetch from the local memory is required, to bring subsequent pattern characters into the PEs. The contents of the ETI regesters are to interpreted similarly.

Galil's algorithm employs local bulletin boards *lbbs* for local communication. Each PE in our machine is endowed with four *lbb* registers, each 16 bits in length. The *plbb* and *tlbb* point to ones within the pattern and text portion of the $SWITCH$ array. During a regular step, the contents of

"adjacent" bulletin boards need to be consulted. We include two registers, the *oplbb* and *otlbb* whose purpose is to hold the contents of appropriate neighbor's bulletin boards. An additional bulletin board register, BB, is used for global communication when the algorithm is in the periodic mode. Unlike a PRAM however, our SIMD machine contains no common memory which is accessible by all processors. Instead, whenever the BB register in some processor is updated, we will broadcast the information to all BB registers in the system. A BB register holds two pieces of information: the period size P, and the length of the periodic string L. Recall from chapter three, that $L = l \cdot p$, where l is the number of occurrences of the period P which have been discovered. We have assumed that the pattern length is one half the text length. Since our PSP is being designed to handle problem instances up to 2^{32} characters, 24 bits suffice to hold L and an additional 24 bits are more than adequate to hold the maximum period size, P. The DRR register is employed to route BB and various *lbb* register values through the system as needed.

Galil's algorithm commences by searching through Z to discover all indices where prefixes of the pattern of length two begin. The algorithm will write one's into all locations r in $SWITCH$ where these prefixes, denoted by $X^{(1)}$ have been discovered (refer to chapter three for an explanation of notation). In stage two, the algorithm seeks to ascertain which of the specified locations correspond to instances of $X^{(2)}$, i.e., prefixes of the pattern

of length 2^2 or 4. In general, in stage (i+1), if $X^{(i)}$ is not periodic, an attempt is made to determine which occurrences of $X^{(i)}$ (i.e., prefixes of the pattern of length 2^i) correspond to instances of $X^{(i+1)}$ (prefixes of the pattern of length 2^{i+1}). The general form of $X^{(i+1)}$ is $X^{(i)}Y^{(i)}$. When stage $i+1$ begins, block size is 2^{i-1}. All processors within a given block work in unison to verify that the suffix of $X^{(i+1)}$, i.e., the $Y^{(i)}$ substring, actually occurs at the indicated location in Z. Processor j within this block checks if:

$$X_k = Z_{k+\Delta} \text{ for } k = \{j + 2^i, j + 2^i + 2^{i-1}\}$$

where j refers to the number of a processor within a block (refer to chapter three for a more detailed explanation).

Our machine handles this step somewhat differently. Requisite pattern characters are delivered to appropriate PEs so that they may be compared with the corresponding text character. This is an instance of the Random Access Read (RAR) problem discussed in the literature. See for instance [*THOM*78] or [*NASS*81*a*]. The DRR will be employed in the neccessary broadcasting routines. Up to two pattern characters may be routed to a particular PE, the first would reside in the FPR (first pattern register) and the second in the SPR (second pattern register). A data selector may be used to route the proper pattern register into a Comparator unit where a comparison with a text character (held within the TR) may be made. The performance of our PSP will to a large degree hinge upon the effectiveness

with which the RAR (or data bradcasting) problem can be handled. We outline several solutions in section 6.8.

§ *6.6 A sample problem for the PSP*

But first, it might be instructive to consider a sample problem for the PSP. To simplify the discussion, once again we state our assumption that neither pattern nor text length will exceed 2^{16}. We focus our attention on the string $Z = abaa\$ababababaa$. This sample string served as our first example for Galil's algorithm in chapter three (the reader may with to refer to page 41 at this time). In this example, the pattern $X = abaa$ and the text string Y equals *abababaa*. The control unit would load X into $PR_0 \cdots PR_3$, i.e., the pattern registers of the first four processors. Meanwhile, Y would be copied into the first eight text registers, i.e., $TR_0 \ldots TR_7$. The end of pattern indicator (EPI) in PE_3 and the end of text indicator (ETI) of PE_7 would each be set to 00. The text switches (TS) in $PE_0 \ldots PE_7$ would each be set to 0, as would the pattern switches (PS) in the first four processors. The *plbb*, *tlbb oplbb*, and *otlbb* registers in $PE0 \ldots PE7$ would be initialized to -1. BB is set to 0, indicating no periodicity has been discovered. The FPR and SPR registers may be assumed to have been initialized to values of -1 as well. This initial configuration of the PSP is shown in figure 6.10. Only the registers whose values we have cited are included. Note also, that

only the first 2^3 of the 2^{16} PEs in our machine are depicted. The contents of the PE Address Registers (PEARs) for processors 0 through 7 is also displayed.

Galil's algorithm dictates that $\lceil \log n \rceil$ stages be performed. We may assume that the control unit is aware of pattern and text length when the string Z is loaded. The first stage of the algorithm searches for all indices j within Z where prefixes of the pattern of length two begin, i.e., for all occurences of $X^{(1)} = ab$. Therefore, the first pattern character, $p_1 = a$, must be broadcast to the first pattern register (FPR) of $PE_0 \ldots PE_6$, while p_2 which is a b must appear in the second pattern register (SPR) of $PE_1 \ldots PE_7$. Obviously a pattern of length 4 cannot occur past location 5 within the text string. However, we follow Galil's convention and engage all 8 processors in the processing. Modifying this algorithm to find longest prefixes of X in Z is a straightforward exercise. The broadcast component of stage one entails two phases. In phase one the a in the PR (pattern register) of PE_0 will be transferred to the DRR (data route register) of this same processor. Next, DRR(0) will broadcast its contents to DRR(1)...DRR(6). Then, these seven routing registers transfer their contents to their own FPRs. The appropiate code is given below:

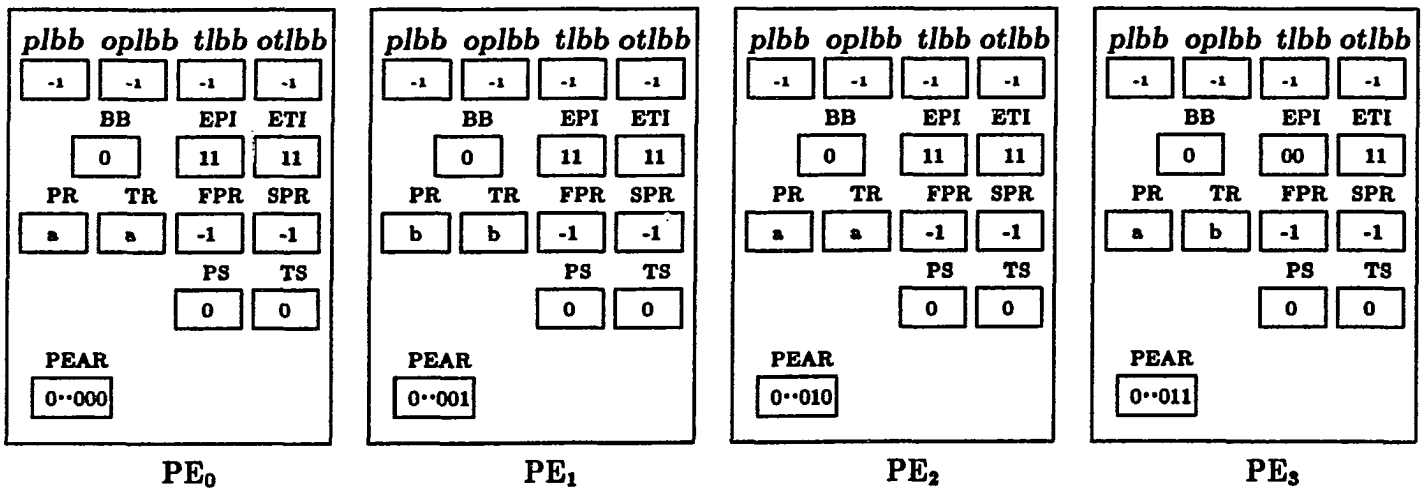


Figure 6.10: Initial Configuration for the PSP when $Z = abaa\$ababababaa$.

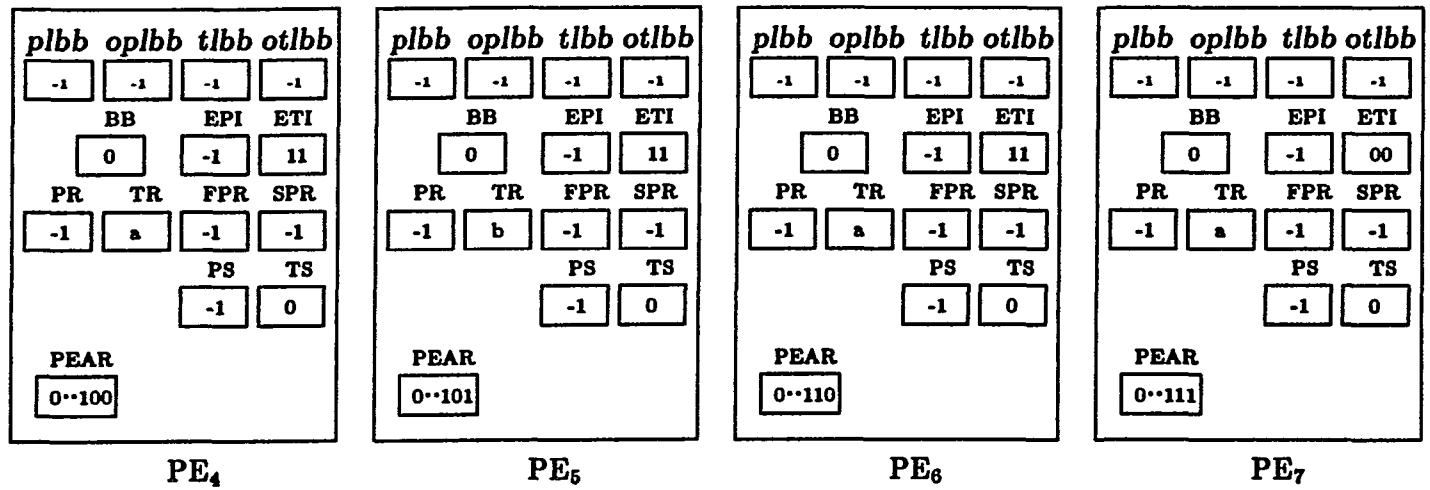


Figure 6.10: Continued.

1. DRR(0) $\leftarrow$ PR(0)
2. $j \leftarrow 2n-1$ // j will maintain this value
3. **if** PEAR(k) $\leq j$ **then** throughout the algorithm //
4. BROADCAST from DRR(0) to DRR(k)
5. **for** $i \leftarrow 0$ **to** j **pardo**
6. FPR(i) $\leftarrow$ DRR(i)

To understand how the processors execute this code, we reconsider figure 6.9. The 82 gates depicted are present in each of the 2^{16} PEs in the PSP. These gates control the flow of data throughout each processor. For example, gates 32 and 33 mediate register transfers from the DRR to each of the PR and TR respectively. Gates 8, 9, and 10 control various inputs to the adder while gates 47 and 55 mediate inputs to the logic unit. Note, that for the sake of clarity we have drawn the logic unit and adder as separate components whereas in reality they would be housed within the same unit. The function of each gate is specified below. The numbers refer to those depicted in figure 6.9.

Adder Inputs

Gates 1 to 11 are for left input, gate 12 to 24 for right input.

1. +1 to adder – used to add 1 to a register.

2. -1 to adder – used to subtract 1 from a register, also used to initialize various registers.

3. 00 to adder – used to initialize various registers to 0, e.g. TS. Also used to set ETI registers to 00.

4. 01 to adder – used to set ETI registers.

5. 11 to adder – also used to set ETI registers.

6. 65,536 to adder – used to add 2^{16} to TPR whenever a portion of the text resides in local memory (required for proper indexing).

7. PS to adder – used to route PS to adder for initialization puposes.

8. PPR to adder – used when 2^{16} must be added to this register.

9. AC to adder – employed when BB must be updated.

10. PEAR to adder – useful for change of PE address.

11. EPI to adder – for initializing this register.

12. +1 to adder – see 1. +1 may be used as a right or left input.

13. -1 to adder – see 2. -1 may be a right or left input.

14. 00 to adder – see 3. 00 may be a right or left input. Here it is used to initialize PS or to set EPI.

15. 01 to adder – see 4. 01 may be a right or left input. Used to set EPI register.

16. 11 to adder – see 5. Also used to set EPI register.

17. 65,536 to adder – see 6. Used to add 2^{16} to PPR wherever a portion of the pattern resides in local memory.

18. TS to adder – used to route TS to adder for initialization purposes.

19. AC to adder – used when PEAR contents must be altered, i.e. during a cyclic shift of addresses.

20. SSP to adder – used to increment or decrement second stack pointer.

21. FSP to adder – used to increment or decrement first stack pointer.

22. TPR to adder – used when 2^{16} must be added to this register.

23. BB to adder – used when BB must be updated.

24. ETI to adder – for initializing this register.

Register – to – Register Transfers

25. SSP to LMAR – used to fetch or to replace top of second stack (i.e. text portion of local memory).

26. FSP to LMAR – used to fetch or to replace top of first stack (i.e. pattern portion of local memory).

27. LMBR to DRR – bringing a new pattern or text character "into the system".

28. IR to FSP – used to initialize this pattern stack pointer (once DMAC is through loading local memory).

29. IR to SSP – used to initialize this text stack pointer.

30. IR to DRR – to activate DRR whenever a BROADCAST routine is required.

31. DRR to IR – used to notify IR when BROADCAST is complete.

32. DRR to PR – to store a processor's pattern character.

33. DRR to TR – to store a text character.

34. DRR to *plbb* – to save index within pattern string where a prefix of pattern commences (within this block).

35. DRR to *tlbb* – to save index within text string where a prefix of pattern commences (within this block).

36. DRR to TEST – used to store a constant (or pair of constants); useful in masking.

37. DRR to AC – used in address changes.

38. DRR to *otlbb* – to save index within text string where a prefix of pattern commences (in a neighboring block).

39. DRR to *oplbb* – to save index within pattern string where a prefix of pattern commences (in a neighboring block).

40. DRR to SPR – to hold a pattern character for comparison purposes.

41. DRR to FPR – to hold a pattern character for comparison purposes.

42. PR to DRR – to prepare a pattern character to be BROADCAST.

43. PR to OVR – when pattern length exceeds 2^{16}, pattern characters will need to be written back into memory.

44. TR to OVR – when text length exceeds 2^{16}, text characters will need to be written into memory.

Inputs to Logical Unit

Gates 45 to 52 are for left inputs; 53 to 60 are for right inputs.

45. PR to logic unit – for comparisons with FPR and SPR characters.

46. TR to logic unit – also for comparisons with FPR and SPR characters.

47. *plbb* to logic unit – for comparison with *oplbb* register (i.e. plbb for neighboring block).

48. *tlbb* to logic unit – for comparison with *otlbb* register.

49. TEST to logic unit – used to enable or disable a PE.

50. EPI to logic unit – to indicate when subsequent fetches from memory are mandated. Also to demarcate end of pattern.

51. PEAR to logic unit – to control address masks.

52. PS to logic unit – to check contents in pattern portion of SWITCH.

53. FPR to logic unit – for comparison with characters contained in PR and TR.

54. SPR to logic unit – also used for comparison with characters contained in PR and TR.

55. *oplbb* to logic unit – for comparison with *plbb*.

56. *otlbb* to logic unit – for comparison with *tlbb*.

57. TEST to logic unit – see 49 above - used as a right entry into logic unit here.

58. TS to logic unit – to check contents in text portion of SWITCH.

59. BB to logic unit – to check if algorithm is in the periodic loop.

60. ETI to logic unit – to demarcate end of text. Also will indicate if subsequent fetches from memory are necessary.

61. *plbb* to DRR – used to broadcast pattern bulletin board contents to a neighboring block.

62. *tlbb* to DRR – used to broadcast text bulletin board contents to a neighboring block.

63. *oplbb* to BB(0:23) — used to trasfer period size to BB when periodicity is discovered.

Logic Unit Outputs

64. output to PS – will indicate the result of comparisons within pattern segment of string.

65. output to TS – will indicate the result of comparisons within text segment of string.

66. output to CHECK register – to specify the result of some test made.

Adder Output

67. output to EPI – used for initializing this register.

68. output to PEAR – used when address register must be changed.

69. output to PPR – used when pattern pointer register must be modified.

70. output to PS – used in initializing PS register.

71. output to TS – used in initializing TS register.

72. output to *tlbb* – used to set *tlbb* register to point to one's in text portion of SWITCH array.

73. output to *lbb* – used to set *plbb* register to point to one's in pattern portion of SWITCH array.

74. output to LMBR – used to store PS and TS values when pattern and text lengths exceed 2^{16}. Once again we remind the reader that memory would have to be augmented suitably to hold this additional information.

75. output to FSP – used for adding +1 or -1 to FSP when pushing or popping pattern stack portion of local memory.

76. output to SSP – used for adding +1 or -1 to SSP when pushing or popping text stack portion of local memory.

77. output to TPR – used when text pointer register needs to be modified.

78. output to BB – used for initializing and updating BB.

79. output to ETI – used for initialization purposes.

Reading and Writing from Local Memory

80. OVR to LMBR – loading the buffer register with either a pattern or text character which needs to be stored. (when these respective strings exceed 2^{16} characters).

81. LMBR to memory – used to store the LMBR into the memory word specified by the MAR.

82. memory to LMBR – used to load the LMBR from the memory word specified by the MAR.

In an SIMD machine, instructions are decoded by the main control unit. We assume that in each instruction cycle, the control unit in the PSP places an instruction (actually a microinstruction) in the IR of each processor. This microinstruction will specify which of the 82 gates within each PE should be opened to activate the appropriate register transfers. Multiple transfers in parallel should certainly be permissible. However, it would not make much sense to have one register loaded from two different sources. For instance, in figure 6.9, if each of gates 80 and 81 were open, would the value in the LMBR which is written into memory correspond to the old value in the LMBR or the new value from the OVR register. To prevent such potential troublespots, the clock cycle within each PE will be divided into three subcycles. During the first subcycle, gates 1 though 63

may be opened, during the second subcycle gates 64 through 80 may be activated, and during subcycle three one of gates 81 and 82 may be opened.

We return at this point to the code on page 167. It is reproduced below for the sake of convenience. Comments concerning the data paths entailed are also provided.

1. DRR(0) ←PR(0)	// during cycle 1 : gate 51 is opened. Each PEAR is tested against zero. In the second subcycle of cycle 1, gate 66 would be opened. During cycle 2 only in PE_0 would the transfer of a pattern character to the DRR take place (gate 42 in PE_0 opened). //
2. $j \leftarrow 2n-1$	// In cycle 1 for this instruction gate 30 would be opened, transferring the value of j into the DRR (we assume that the control unit places this value into the IR).//
3 if PEAR(k) $\leq j$ **then**	// In the 1^{st} subcycle of cycle one, gates 51 and 57 would be open. The comparison unit receives a copy of this PE's address and the value of j from the TEST register. In the 2^{nd} subcycle, gate 66 would be opened, putting the result of the comparison into the CHECK register (1 $\equiv PE\ address \leq j$). //
4. BROADCAST from DRR(0) to DRR(k)	// i.e. BROADCAST to all processors where CHECK register = 1. This "system macro" will be explained shortly. //

5. for $i = 0$ to $j-1$ pardo	// merely have a microinstruction which checks the CHECK register. if CHECK = 1 then proceed to 6. below. //
6. FPR(i) ← DRR(i)	// when BROADCAST is complete, the requisite pattern character is in the DRR. Hence opening gate 41 will accomplish this transfer. //

When this phase of the required routing is done, the second component of $X^{(1)}$, i.e. the b , will be routed to the second pattern registers (SPRs) of PE_1 to PE_7. The broadcasting entailed in phase two would be accomplished by the following routine:

DRR ←PR(1)	// PR(1) holds p_2, the second pattern character. In cycle one gates 51 and 66 would be opened. For PE_0 alone, the CHECK register would = 1. Next would be a test microinstruction for this CHECK register. And in PE_1, gate 42 would be opened, placing p_2 which is a b into DRR(1). //
if 1 ≤ PEAR(k) ≤ j then	// open gates 51, 57 and 66. Note we assume the comparison unit can check if j ≥1. If CHECK = 1 then notify control unit of participation in this BROADCAST. //
BROADCAST from DRR(1) to DRR(k)	// circulate the b to appropriate PEs //
for i = 1 to j pardo	// similiar to previous *if* statement. //

SPR(i) ←DRR(i) // if CHECK = 1 in a PE, then gate 40 is opened, enabling this transfer. //

BROADCAST is a routine which has two sets of parameters. The first being a source list; in the segment of code above, this list is just the single register, DRR(1). The second parameter is a destination list, in the code above this list is equal to DRR(1) to DRR(7). It is assumed that each register in the destination list will receive data from just one register in the source list. Let the source list consist of $s_1 s_2$ and the destination list be equal to $d_1 d_2 d_3 d_4$. Then s_1 will be routed to each of d_1 and d_3, and both d_2 and d_4 will receive their data from s_2. This convention corresponds to the manner in which prefixes of the pattern must be distributed throughout the Z string. There are several alternative methods of executing a BROADCAST instruction, these will be explored shortly.

The data routing for stage one is complete. Pattern characters are located in the appropriate processors so that the requisite comparisons to identify prefixes of length two within Z may be made. A regular step may consist of four rather than two sets of comparisons. This is due to the format we have chosen for entering pattern and text characters within PEs. The two registers specified by a COMPARE instruction will each load their contents into the comparison unit (see figure 6.9) which will place a 1 in the pattern SWITCH (PS) or text SWITCH (TS) if the two characters match and a 0 otherwise. The code is given below.

PS ←COMPARE(FPR, PR)	// Opening gates 45, 53, and 64 will accomplish this task. //
IF PS = 1 THEN PS ←COMPARE(SPR, PR)	// It would be senseless to make this second comparison unless the first had succeeded. First have a microinstruction to test the PS register. Within any PE where PS = 1, gates 45, 54 and 64 in the next PE should be opened. //
TS ←COMPARE(FPR, TR)	// Open gates 46, 53, and 65. //
IF TS = 1 THEN TS ←COMPARE(SPR, TR)	// Test the TS register. If TS = 1 within a PE, then gates 46, 54, and 65 in the next PE should be opened. //

And finally:

IF PS(j) = 1 THEN plbb(j) ←PEAR(j)	// Test each PS register. If PS(j) = 1 then open gates 10, 14, and 73 in PE_j. //
IF TS(j) = 1 THEN tlbb(j) ←PEAR(j)	// Test each TS register. If TS(j) = 1 then open gates 10, 14, and 74 in PE_j. //

The contents of the cited registers at the close of stage one are shown in figure 6.11. The PR, TR, EPI and ETI registers appear as they did in figure 6.10. In fact, these registers remain unchanged until the algorithm is done. The first round of broadcasting is complete, hence the prefix *ab* of pattern is located in the FPRs and SPRs of the appropriate PEs. The BB, *oplbb*, and *otlbb* registers have not been updated and therefore still contain -1 entries. $plbb_0$ contains a 0 and *tlbbs* 0, 2, and 4 contain entries

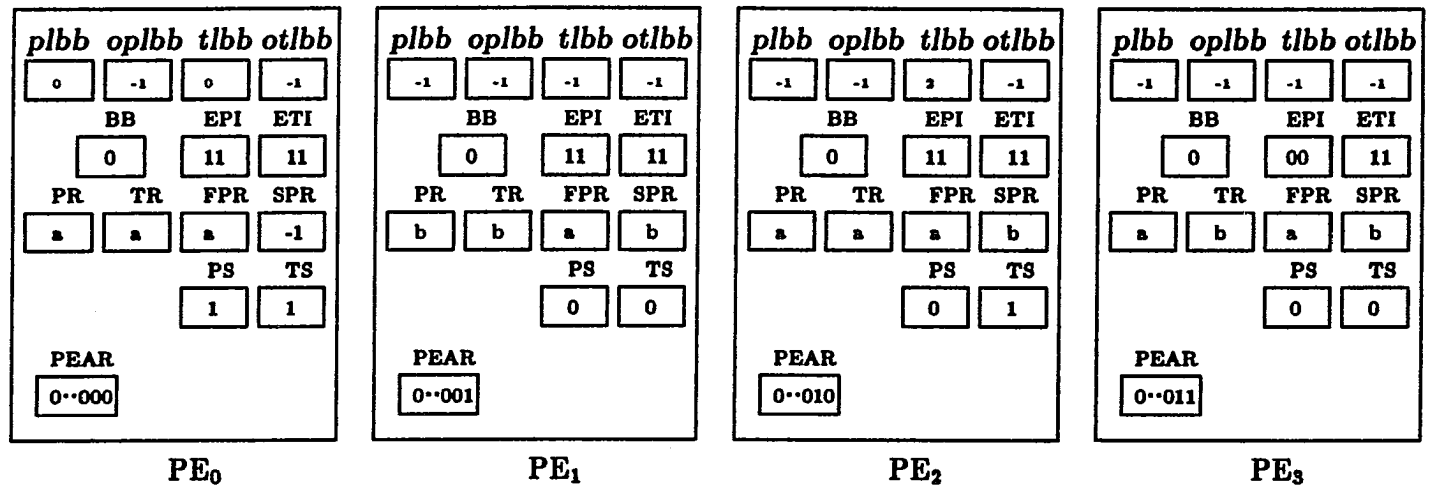


Figure 6.11: Contents of relevant PSP registers at the end of Stage one.

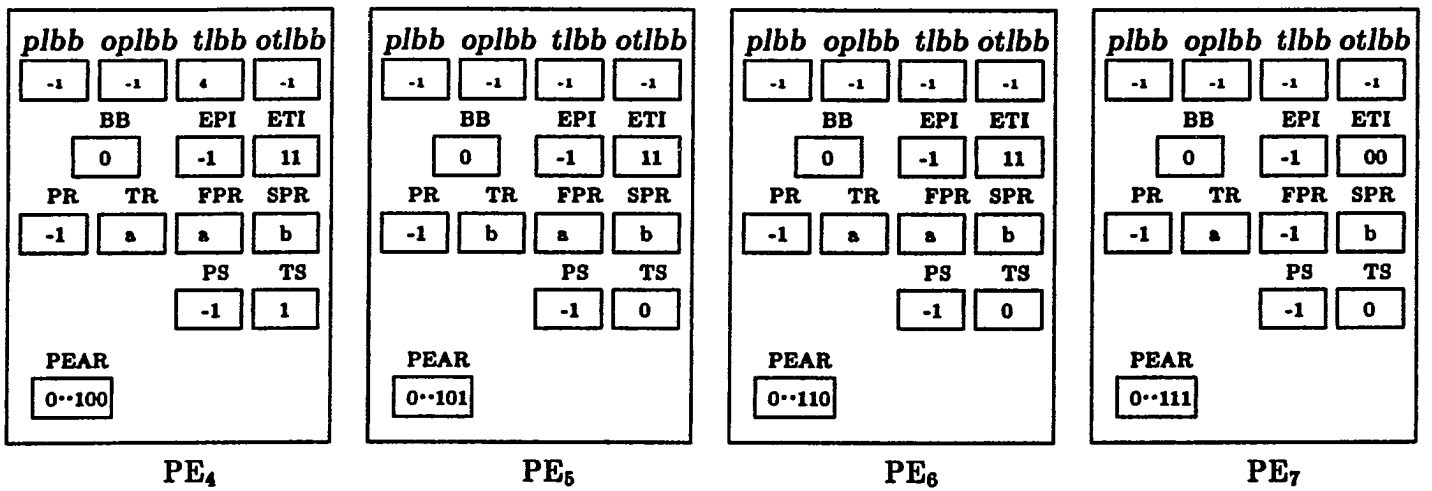


Figure 6.11: Continued.

of 5, 7, and 9 respectively. We have numbered the entries in the *tlbbs* in a manner closer to Galil's original scheme. The reader may wish to compare figure 6.11 with the information contained in the arrays on page 41.

We are ready to begin stage two. The discussion here will parallel that given on page 42. $i+1=2$, hence the block size equals 2^{i-1} which is 2^{1-1} or $2^{\circ}=1$. The *plbbs* in PE_0 and PE_1 need to communicate:

DRR(1) ←*plbb(1)*	// In cycle one, open gates 51, 66. Comparator checks PE address. In PE_1, CHECK = 1 and in cycle 2 gate 61 would be opened within this processor. //
BROADCAST(DRR(1) to DRR(0))	// PLBB(1) register contents are routed to PE_0 //
OPLBB(0) ←**DRR(0)**	// In PE_0 only, gate 39 would be opened. //

Next, the contents of *plbb(0)* and *oplbb(0)* are compared. If a non-negative entry appears in each register, then the algorithm must enter the periodic mode. The period size P, and L which is the length of the periodic string, would be written into BB(0). This information would then be broadcast throughout the PE ensemble. The appropriate code is given below.

COMPARE POSITIVE (PLBB(0), OPLBB(0))	// We assume these two checks can be made at the same time by the Comparison Unit. Gates 47, 55, and 66 in PE_0 would be opened. If both registers are non-negative a '1' is written into CHECK.//
IF CHECK = 1 THEN BB_0[0:23] ←OPLBB BB_0[24:47] ← OPLBB BROADCAST(BB(0) to BB(1) ... BB(7))	// if CHECK = 1 then periodicity has been found. Only in PE_0 would gates 1, 23, 63 and 78 be opened. The period size which is contained in *oplbb(0)* is gated to the left half of BB(0) and also to the right half of BB(0). Note that it may prove more convenient to place two BB registers per PE, one to hold period size, P, and a separate BB register for L. Also, BB(0) could be routed to DRR(0) in two cycles using the Adder and LMBR as intermediate destinations. //

In the example we have been describing, the CHECK register in PE_0 would contain a 0, indicating that a regular step is to be performed in stage two as well. The prefix $X^{(2)} = abaa$ would be routed to those locations where searches for occurrences of $X^{(1)} = ab$ have been successful. Referring to page 41 and figure 6.11, it is evident that copies of the substring *aa* must be *shipped* to locations Z_3Z_4, Z_8Z_9, $Z_{10}Z_{11}$ and $Z_{12}Z_{13}$. There is no *overlap conflict* in these locations and hence only the FPR's wil be required. The third pattern character, p_2 (for convenience we start indexing with p_0 in this chapter) which is an *a* will be broadcast to PE_2, PE_4, and PE_6. Whereas the fourth pattern character, p_3, also an *a* is routed to each of PE_3, PE_5, and PE_7. The following code would accomplish the requisite routing. The

reader may wish to refer to figure 6.9 to determine proper gate sequences.

set: f $\leftarrow 2^{count-1}$, *g* $\leftarrow 2^{count} - 1$	// *COUNT* is a variable that indicates algorithm stage numbers. It is maintained by the Main Processing unit. f and g will be lower and upper bounds respectively on the index of characters which must be broadcast (essentially, they demarcate the $Y^{(i)}$ segment which needs to be checked for. //
for m = f to g pardo DRR(m) $\leftarrow$ **PR(m)**	// BROADCAST(DRR(f) ... DRR(g) to DRR(2)DRR3; DRR(4)DRR(5); DRR(6)DRR(7) // A check would be made of all location where (*plbb* or *tlbb*) $\neq -1$ //
for k = 2 to 7 pardo FPR(k) $\leftarrow$ **DRR(k)**	// In general, this transfer takes place within all PEs, where the *plbb* or *tlbb* is not equal to -1. //

Next, comparisons between FPR contents and appropriate PR and TR registers must be made. Results of these comparisons are written into the corresponding PS and TS registers. The details entailed are similiar to the description for this process in stage one (see page 41). The contents of relevant registers at the end of stage two are illustrated in figure 6.12. Notice that the *tlbb* regesters in PE_0 and PE_2 have been reset to -1. This is because at locations 0 and 2 within the text string, occurrences of $X^{(1)} = ab$ were discovered in stage two, not to been followed by the requisite suffix $Y^{(1)} = aa$. Only the 4 in *tlbb(4)* survived, indicating that one occurrence of the pattern X = *abaa* was found, commencing at position 4 within the

Figure 6.12: Contents of relevant PSP registers after Stage two.

Figure 6.12: Continued.

text string (recall that we began our indexing with position 0 here). The reader may wish to compare figure 6.12 with the SWITCH array provided on the top of page 44 before proceeding. Finally, the results contained within the first 8 TS registers (and in general, within all TS registers up to and including the one residing in the PE whose ETI contents equal 00) would be transferred to the LMBR registers and then a WRITE into the local memories would be initiated. If desired, a 1 may be added to the address of each TS, to maintain consistent indexing. When the set of occurrences of the pattern within the text is known to be somewhat sparse it may be desirable to WRITE those TPR values where corresponding TS registers contain 1's rather than all TS values in the system. Once the WRITEs are complete, the DMAC would mediate appropriate output transfers. Notice that it is not necessary to output the contents of the PS registers.

In chapter three we considered a second string matching example which caused the algorithm to enter the periodic mode. We spare the reader the intricate details of the PSP program, and instead highlight several noteworthy features. The first step in every stage of Galil's algorithm is to check if the string $X^{(i)}$ is periodic. In chapter three, we observed that this chore was accomplished by the *lbb* in the 2^{nd} block. However, as we saw in the prior example, the *plbb* in the second block would be routed to the *oplbb* register in PE_0 where the requisite calculations would take place. If

the contents of *oplbb(0)* are positive, then its value, which corresponds to the period size P, would be transferred into the left half of BB(0).

If $X^{(i)}$ is periodic, then the algorithm proceeds to BOX 5 (see figure 3.1) where a test is made to determine if the periodicity continues. To determine if the string $\hat{X}^{(i+1)}$, which was defined (on page 45) as the prefix of X of size $2^i + L$, is periodic or not, a check must be made of SWITCH values at positions P+1 and L+1. In the PSP, the requisite PS values (from PE_P and PE_L would be routed to the TEST register in PE_0 where the comparisons would be made. If it is discovered that $\hat{X}^{(i+1)}$ is periodic then the value of L must be updated. As described on page 48, if $2L - P > 2^{i+1}$ then L $\leftarrow 2L - P$, otherwise L is set to 2L. We have not included a subtractor in our PE ensemble. One way to circumvent this apparent difficulty is simply to include a 1's complement capability on each input of the adder. A second solution is just to compare 2L against $2^{i+1} + P$. Each of these strategies would entail adding an extra data path (and gate) from BB(0) to the left adder input. The third and perhaps least cumbersome approach is to just let the main processor perform the requisite subtraction *and* multiplication. Each of L and P, which are present in BB(0), would be routed to IR(0). This transfer is somewhat clumsy in our machine, entailing intermediate "stop overs" in the adder, LMBR, and DRR. The latter two registers would have to be suitably expanded to handle BB contents. Note that COUNT, the variable which monitors stage numbers is also present in the main processor.

In any case, once a firm decision has been made on which solution is optimal, extra gates and/or registers could be added to our preliminary design.

We have been discussing the strategy within the PSP for checking if the string $\hat{X}^{(i+1)}$ is periodic, and for updating L appropriately when the outcome is affirmative. Naturally, the second possibility that may ensue when this test is conducted, is that the periodicity within $X^{(i)}$ does not continue into $\hat{X}^{(i+1)}$. In this case, a search must be undertaken to locate the unique important occurrence present within the first $L+1-P$ positions of Z. Recall, that an occurrence of v at j is said to be important if v does not occur at j+P. In the PSP, this search would be accomplished by checking values of the PS registers within the first block of the PE melange. When the index of the important occurrence has been detected, this information would be routed to BB(0). A subsequent BROADCAST from DRR(0) to appropriate PEs in the other blocks (i.e. to those PEs containing 1's within either their PS or TS registers) would ensue. This would be followed by a "pruning procedure" to rid the PS and TS registers of "superfluous" 1's (this step corresponds to BOX 7 in the flowchart for Galil's algorithm). Special occurrences (consult chapter 3 for definition) would be handled in a similar manner. The next step would be to execute the regular step depicted in BOX 4. Our previous example has already touched on the details involved.

§ *6.7 Microprogramming format of the PSP*

There are two types of microinstructions to be executed by the PSP. They are the TEST and GATE instructions. There are several registers which require occasional testing. Firstly, the BB register must be compared to zero, to indicate whether the algorithm is in a regular or periodic mode. It is the *plbb* and *oplbb* registers which must be consulted to ascertain period size. And in the case where periodicity terminates, these registers will aid in the search for important and special occurrences. When periodicity terminates, it is necessary to access the *tlbb* and *otlbb* registers as well. Their contents will help the PSP to ensure that blocks of SWITCH within the text portion of Z match the initial block of SWITCH in the pattern string. It is also necessary to check the EPI and ETI registers to establish proper demarcation addresses to delineate the active portion of the PE ensemble. To establish source and destination lists for a BROADCAST instruction, it is necessary to inspect PEAR registers. And finally of course, the PS and TS registers themselves will need to be tested. For the most part, we have conducted the requisite testing with the aid of the comparison units, results appearing in the CHECK registers. A TEST instruction would consist of a one bit opcode, where 0 would denote a TEST instruction, and a comparison field which would indicate the value that CHECK is being tested against. Since the PSP is an SIMD machine, rather than specifying a jump address within a TEST microinstruction, PEs which "fail" a

test would merely be disabled during the execution of a subsequent instruction. The CHECK register is thereby being employed as a form of mask register.

A GATE microinstruction will have an opcode of 1. In a pure horizontal approach to microprogramming, individual bits would directly specify which of the 82 gates depicted in figure 6.9 are to be opened during the execution of an instruction. This would necessitate a microword size of 83 bits. The format of a horizontal GATE microinstruction appears in figure 6.13.

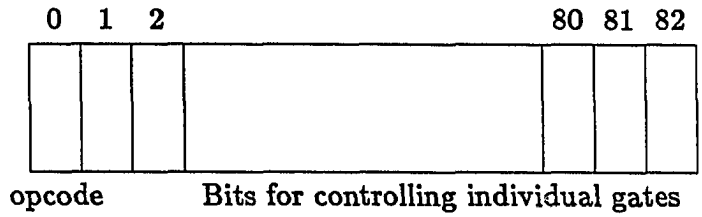


Figure 6.13: Format for the GATE microinstruction in the PSP if a horizontal organization is employed.

The micromemory for the PSP is stored within the main control unit. A microprogram counter is employed as an address register for fetching words from the micromemory, and a microinstruction register serves as a memory buffer register to hold individual instructions. If a purely horizontal approach were adapted, then the size of the MIR and the size of every word in the micromemory would of necessity be 83 bits.

The GATE microinstruction format depicted in figure 6.13 has the capability of specifying 2^{82} -1 register transfers. Perusing figure 6.9 once again should convince the reader that very few of these potential instructions would have a meaningful interpretation. For instance, the adder unit may have only one left input, one right input, and one output path. This suggests employing 4 bits to encode each of gates 1 through 11, gates 12 through 24, and the outputs on lines 67 to 79. Similarly, 3 bits may be used to encode each of the logic unit's left and right inputs (gates 45 through 52, and gates 53 through 60 respectively). Two bits suffice to encode the outputs of this unit. There are 16 data paths which either lead in to or out of the DRR register. Also gates 25, 26, 28, 29, and 80 are unaccounted for. We employ 5 bits to designate these paths. And finally, we individually control gates 81 and 82 of the LMBR with one bit for each line. A GATE microinstruction in this essentially vertical approach would appear as in figure 6.14.

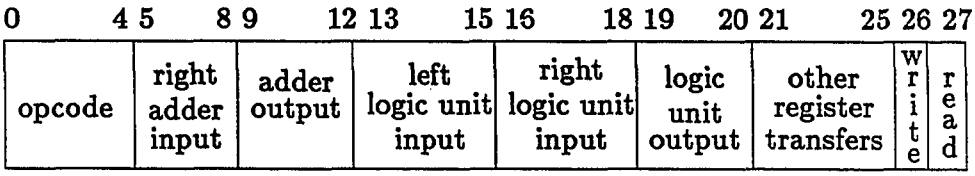


Figure 6.14: Format for the GATE microinstruction in the PSP if a vertical organization is employed.

Notice that microword size has been reduced from 83 bits to 28 bits.

When encoded subfields are utilized, as in the vertical approach just described, it is of course necessary to provide for decoding. Encoded subfields would appear on the input of each decoder and individual gate control signals would be present on the output lines. In a serial computer, the price tag for this vertical approach would be 7 decoders. One 2 × 4 unit, two 3 ×8 devices, three 4 ×16 units and one 5 ×32 decoder would be required. The savings in micromemory word size would certainly warrant such an expenditure. However, in the PSP, the seven decoders would have to be replicated in each of the 2^{16} PEs of the system. The resulting economy in micromemory word size (there is only one micromemory – it is present in the main control unit) would certainly not by itself justify an expenditure for 7 $\times 2^{16}$ decoder units. However, micromemory word size is not our chief concern. During each instruction cycle, an instruction (i.e. a microinstruction must be BROADCAST from the main control unit to the IR in every PE of the system. The duration of this instruction BROADCAST phase will play a fundamental role in determining the system's instruction cycle time and hence its potential operation speed. The benefits accrued from reducing the length of the instruction BROADCAST from 83 bits for a horizontal approach down to 28 bits if a vertical methodology is adapted, may well offset the requisite decoder expenditure. Again we cite the need for simulation studies.

The subject of data paths also merits some discussion. In figure 6.9

individual registers possess dedicated data paths to other registers. In a processor with n registers, it is possible that $O(n^2)$ such paths may be required. The "spaghetti-like" profusion of data paths which may ensue can be remedied by employing data buses rather than dedicated data paths to mediate register to register transfers. Four bus lines would serve as left and right inputs to each of the adder and logic unit, and an additional two lines would be necessary to route their respective outputs. For a lucid exposition on the effect that utilizing bus lines would have on the microprogramming level organization, the reader is referred to [TAN76].

Before discussing the format of the BROADCAST "macro" one final comment is in order. Our original design for the PSP assumed that text length was precisely equal to twice the pattern length. At the end of chapter three, we related Galil's strategy for removing this restriction. When pattern and text lengths are unrelated, the text is to be divided into overlapping segments and the processors are to search for occurrences within these segments. This approach may necessitate more frequent accesses to the local memories. Distribution of text and pattern characters would be somewhat affected as of course would requisite BROADCAST patterns. However, with minor modifications the basic PSP design would remain intact.

§ *6.8 Approaches to the BROADCAST problem*

We have deferred the details entailed in the BROADCAST routine we made use of earlier until this time. We shall briefly discuss each of the approaches that may be employed and cite their respective complexities. We will see that there is no clearcut best approach. Instead, here is an additional instance where simulation studies and further research are needed. Before commencing though, we will once again state the problem. SWITCH in Galil's basic algorithm is an array of $3n+1$ entries. And recall that the format of Z, the input string is $X\$Y$ where X is the pattern and Y is the text string. When the algorithm is done, a 1 in SWITCH at position j will specify an occurrence of the pattern X within the string Z. After stage i, with $1 \leq i < \lceil \log N \rceil$, a 1 at position j in SWITCH signals as occurrence of a prefix of the pattern of length 2^i (roughly speaking). This prefix is denoted by $X^{(i)}$. In stage $i+1$, an attempt is made to determine if those prefixes of length 2^i within Z correspond to prefixes of length 2^{i+1} denoted $X^{(i+1)}$. The general form of $X^{(i+1)}$ is $X^{(i)}Y^{(i)}$. In stage $i+1$, it is precisely this substring, $Y^{(i)}$ which must be broadcast to the appropriate processors in the PSP. Each PE in the system contains a single pattern character which is stored in its PR (pattern register). Therefore, as stage $i+1$ begins, the string $Y^{(i)}$ is contained within PR(2^i), PR(2^i+1), ..., PR($2^{i+1}-1$). The contents of these registers must be broadcast to the FPRs (first pattern registers) and where necessary the SPRs (second pattern registers) of

$PE(j + 2^i)$, $PE(j + 2^i + 1)$,..., $PE(j + 2^{i+1} - 1)$. The DRRs (data route registers) and the perfect shuffle interconnection network are employed in the requisite data movements.

One possible solution for the data broadcasting problem is to use a generalized connection network (GCN). A GCN is essentially a switching network with N inputs and N outputs which is capable of realizing any mapping of inputs onto outputs. Thompson in [THOM78] explains that any GCN construction leads automatically to an algorithm for the transfer of data among PEs in an SIMD computer. To simulate Thompson's GCN network requires $8 \log N$ routing steps on a PSC (perfect shuffle computer). The setting of a GCN may represented by a sequence of N integers, one for each output vertex: $o_0, o_1, \ldots, o_{n-1}$ where $o_k = i \;\; iff$ output number k is connected to input number i. Thompson notes that each output is connected to precisely one input. For example, if $N = 4$ a possible GCN setting might be $(0, 1, 0, 1)$. Input 0 is connected to outputs 0 and 2 while input 1 is connected to outputs 1 and 3.

A necessary and sufficient condition for a graph to qualify as a GCN is that it contain a subgraph with the appropriate connectivity for each of the possible N^N possible o_k sequences. Thompson remarks that the edges in these subgraphs correspond to the switches that should be closed to realize each GCN setting. The configuration shown in figure 6.15 may be employed to obtain a GCN [OFMAN65].

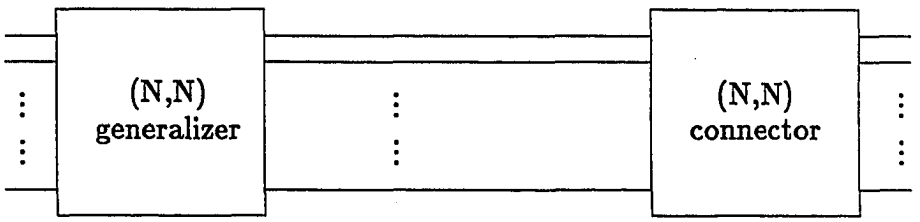


Figure 6.15: Configuration for a GCN construction.

The left-hand network, the generalizer, produces the correct number of copies of each of the inputs. The right-hand network depicted in figure 6.15, the connector is an (N, N)-connection network. An (N, N)-connection network is a switching network with N inputs and N outputs which is capable of performing any of the $N!$ one-to-one mappings of inputs onto outputs. It is the connector which must permute the inputs which it receives from the generalizer to the appropriate output lines. Thompson describes a recursive construction for a connection network which is attributed to Beizer.

The delay for a GCN is the maximum number of contact pairs separating any input-output pair. For Thompson's GCN delay time is $O(\log n)$, however obtaining the proper switch setting for his GCN poses a problem. The switch settings may be found by employing a method of Waksman [$WAK68$]. However, the best known time for Waksman's approach re-

quires $O(n \log n)$ time on a serial computer. Thompson comments that it may be feasible to store precomputed GCN settings. He believes that this approach may require storing one bit per switch setting. As Thompson's GCN has $O(N \log N)$ switches this would entail storing $O(N \log N)$ bits. The requisite broadcast patterns in the PSP possess a good deal of regularity. In every stage after the first, the $Y^{(i)}$ substring which needs to be distributed in stage $i+1$ always begins in PE(2^i) and terminates in PE($2^{i+1}-1$). A similar sort of structure may be found in the destination addresses for these substrings. If as stage $i+1$ begins, there is a 1 at SWITCH(j), then $Y^{(i)}$ needs to be routed to PE($j+2^i$) through PE($j+2^{i+1}-1$). If the 1 at SWITCH(j) "survives" then $Y^{(i+1)}$ will need to be routed to PE($j+2^{i+1}$) through PE($j+2^{i+2}-1$) during stage $i+2$. Of course, if the 1 at SWITCH(j) was turned off during stage $i+1$, then this latter broadcast is unnecessary. The important point, however, is that a list of potential destination addresses for $Y^{(i+1)}$ can always be obtained readily from the list for $Y^{(i+1)}$. It would seem that this prior knowledge should lead to an improved GCN switch setting algorithm *for this application.* Further research is needed here.

Referring once again to figure 6.15, we comment that the generalizer is composed of two portions – a hyperconcentrator and an infrageneralizer. This is depicted in figure 6.16. The hyperconcentrator routes all important inputs to its uppermost output lines. That is, if p of the in-

puts are to appear on some output of the generalizer, they must appear on lines $k_0, k_1, \ldots, k_{p-1}$ of the hyperconcentrator. The infrageneralizer is then responsible for producing the correct number of copies of each of its p inputs. For our purposes, during a regular step in the algorithm, the hyperconcentrator would route $Y^{(i)}$ to lines $k_0, k_1, \ldots, k_{2^i}$. The infrageneralizer would then produce a number of copies of $Y^{(i)}$ which equals the number of 1's in the SWITCH array. This latter quantity could be obtained by the PE melange in $O(\log N)$ time by using the parallel addition algorithm explained in [HILL86].

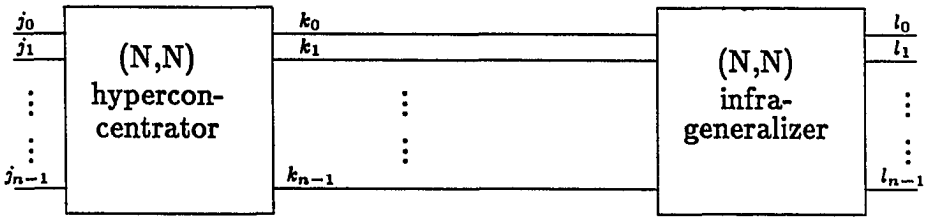


Figure 6.16: Configuration for a generalizer construction.

Nassimi and Sahni [$NASS82a$] comment that the hyperconcentrator and infrageneralizer portions of Thompson's GCN are relatively easy to set up (see [NASS81a]). The connector component of the network may be implemented with a Benes permutation network (see figure 6.17). They provide a parallel version of Waksman's set up algorithm which requires $O(\log^4 N)$ steps on an N PE PSC. Further if $N^{1+1/k}$ PEs are

used, then only $O(k \log^3 N)$ time is needed. Let $D = D(0 : N-1) = (D(0), D(1), \ldots, D(N-1))$ be the permutation which the Benes network is to implement. Input i is to be routed to output $D(i)$ where $0 \leq i < N$. Let the upper half of the Benes network, $B(n-1)$ in figure 6.17, be denoted by $B_u(n-1)$ and the lower portion by $B_l(n-1)$. Nassimi and Sahni's algorithm first obtains the switch settings for those switches in the first and last stages, i.e. stages 0 and $2n-2$ respectively, as well as the permutations D_u and D_l to be performed by $B_u(n-1)$ and $B_l(n-1)$. "The switch settings for the first and last stage and the permutations D_u and D_l are such that if $B_u(n-1)$ realizes D_u and $B_l(n-1)$ realizes D_l, then $B(n)$ realizes D. Since $B(1)$ can realize all permutations of 2 inputs, it will follow by induction that $B_u(n-1)$ can be set up to realize D_u and $B_l(n-1)$ can be set up to realize D_l."[1] Nassimi and Sahni explain that corresponding to the input-output mapping $D(0 : N-1)$, an undirected bipartite multigraph $G(D)$ may be defined. This graph will have vertices $x_0, x_1, \ldots, x_{N/2-1}$ and $y_0, y_1, \ldots, y_{N/2-1}$. Vertex x_i corresponds to the i^{th} switch in stage 0 whereas y_i corresponds to the i^{th} switch in the last stage, i.e. stage $2n-2$. In the mapping, if $D(i) = j$ then the multigraph will contain an undirected edge $(x_{i/2}, y_{j/2})$. These authors comment that Waksman's construction may be viewed as following

[1]D. Nassimi and S. Sahni, "Parallel Algorithms to Set Up the Benes Permutation Network," *IEEE Transactions on Computers*, Vol. C-31, No. 2, February 1982, p. 149.

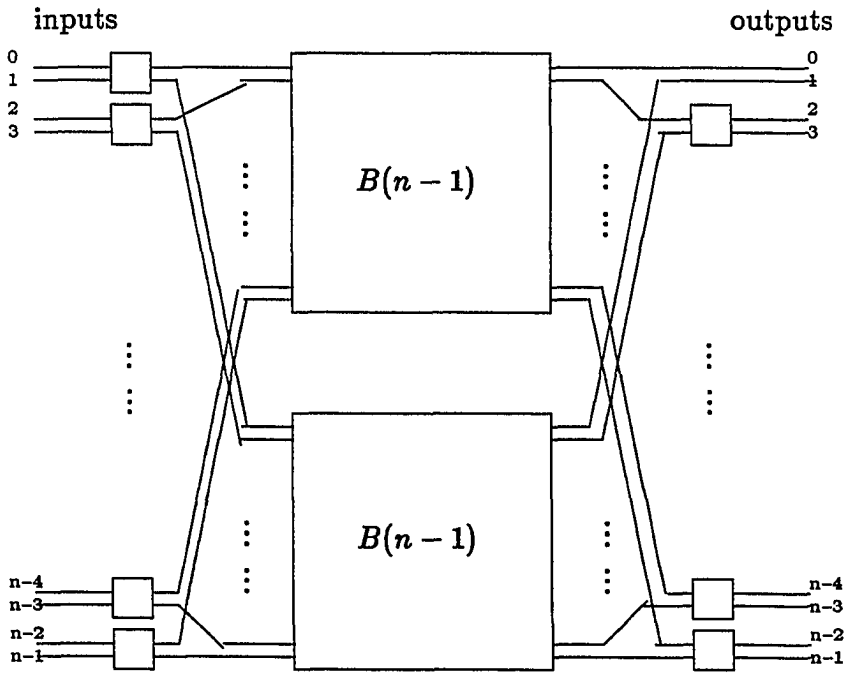


Figure 6.17: The Benes permutation network $B(n), N = 2^n$.

the cycles in the bipartite multigraph $G(D)$ to determine what the state of the switches in the first and last stage of $B(n)$ should be, thus obtaining the permutations D_u and D_l. Essentially, their algorithm finds a complete matching on $G(D)$. An example whereby the matching for $G(D)$ is enlisted to determine the switch settings for a Benes network may be found in [NASS82a].

The bottleneck in employing Thompson's GCN construction to mediate the data transfers in the PSP is the lengthy time required to determine

switch settings in the Benes connection portion of the network. The best parallel algorithm to obtain proper settings, as we cited above, requires $O(\log^4 N)$ time on an N element PSC. It is an open question as to whether the permutations entailed in routing the $Y^{(i)}$ portion of the pattern through out the PE ensemble are amenable to efficient preprocessing.

Another avenue of attack is also suggested by the work of Nassimi and Sahni. In [NASS81b], the authors describe a Benes network which essentially determines its switch settings "on the fly". The total time required for both switch setting *and* transit through the network is $O(\log N)$. This scheme may be adapted to implement permutations efficiently on a PSC as well. The drawback though is that not every permutation is realizable by this procedure.

The authors remark that there have been other permutation networks which have been proposed in the literature that are easier to set up than the Benes network. These other approaches, however, are inferior in terms of delay time or switch requirements. Setting up a full crossbar, for example, is trivial but requires an expenditure of $O(N^2)$ switches. Lang and Stone [LANG76] propose a shuffle-exchange network which is easy to set up but entails a delay of $O(N^{1/2})$. And finally Batcher's sorting network [BATCH68] is self-routing, but has $O(\log^2 N)$ delay and $O(N \log^2 N)$ switches.

In their "self-routing" network, each input will be supplied with a desti-

nation tag which will travel through the system with it. Each switch will be augmented with some simple logic which will enable it to determine its own setting by inspection of its two incoming signals. D_i will denote the destination tag on input terminal i, $0 \leq i \leq N-1$. As before $(D_0, D_1, \ldots, D_{N-1})$ will represent a permutation of $(0, 1, \ldots, N-1)$. The information present on input i is to be routed to output D_i. Nassimi and Sahni show how the switch settings may be obtained from the binary representation of D_i. A Benes network with $N = 2^n$ lines will consist of $2n-1$ stages of switches whose settings need to be determined. The stages are numbered from 0 to $2n-2$. Then the state of a switch in stage b or stage $2n-2-b$ where $0 \leq b \leq n-1$ may be obtained from bit b of the destination tag of its upper input (see figure 6.18).

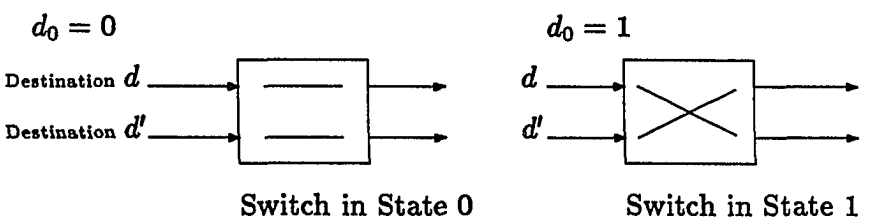


Figure 6.18: Use of destination tags to control a switch at stage b or stage $2n-2-b$, $0 \leq b \leq n-1$.

As we see above, if bit b is 0 then the switch will be set to state 0, otherwise it is set to state 1. Figure 6.19 illustrates the switch settings obtained by this method for $B(2)$. A bit reversal permutation is being performed. That is, input i is being sent to output terminal j where the binary representation of j is the reverse of that for i. A permutation which may not be realized by this scheme is shown in figure 6.20.

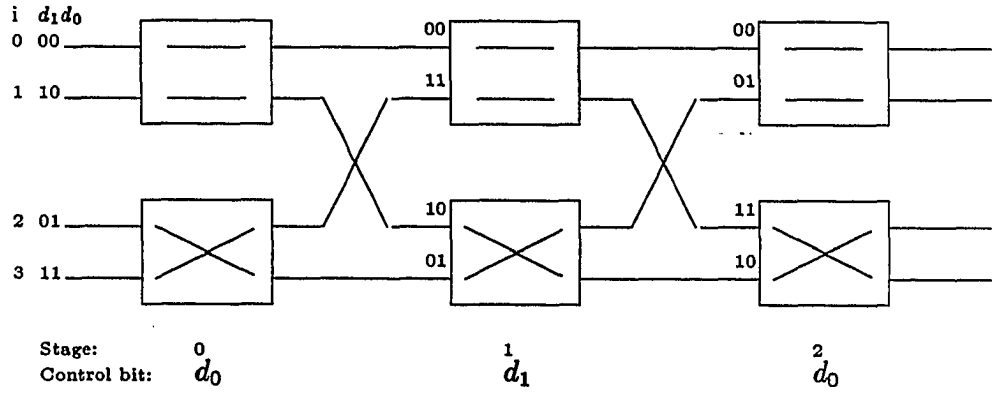


Figure 6.19: Bit Reversal Permutation.

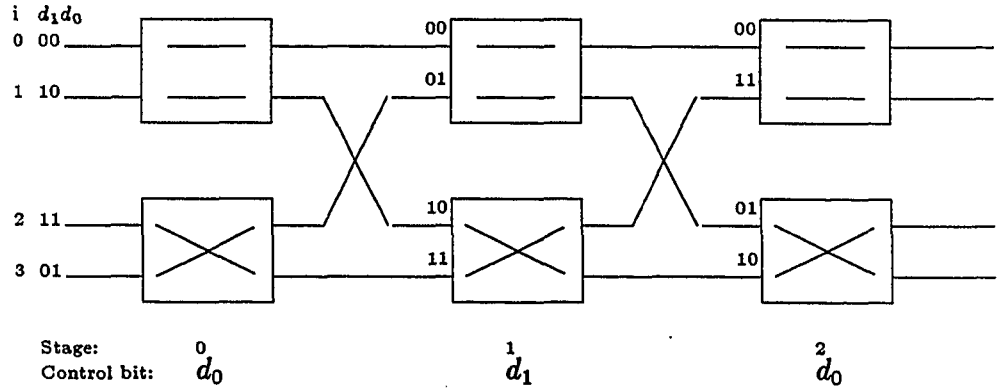


Figure 6.20: A permutation not realizable by this scheme.

This switch setting scheme is similar to that employed by Lawrie [LAWR75] to obtain settings in his omega network. Lawrie's scheme required half as many switches and about half the delay time as this self-routing Benes network. However, the class F of permutations realizable by this Benes network far exceeds those possible by the self-routing omega network. Also, as Nassimi and Sahni observe, it is possible to disable the self-setting logic (when a permutation *not* in F is encountered) and externally determine the switch settings. Such a Benes network can realize all $N!$ permutations. This is not the case for the omega network. The authors also cite earlier work by Lenfant [*LENF*78] who proposed five different set-up algorithms for what he labeled frequently used bijections (FUBs). However these FUBs constitute a small subset of Nassimi and Sahni's set F of realizable permutations.

The authors propose that an SIMD computer could gainfully be endowed with *two* interconnection networks. The first could be a perfect shuffle scheme. The second would be the self-routing Benes network they describe with just $O(\log N)$ delay. PE(i) would be connected to input i and output i of $B(n)$ where $0 \leq i \leq N-1$. Some permutations would be more efficiently performed on the first network, while for others $B(n)$ would exhibit superior performance. Also to be noted, is that when a permutation lies in F, then realizing this mapping via $B(n)$ requires $O(\log N)$ *gate delays.* The authors show how the same permutation may be performed by the perfect shuffle network in $O(\log N)$ *routing steps.* Each routing step, however, requires broadcasting an instruction to each PE in the system. Therefore, performing the permutation directly through $B(n)$ rather than indirectly (via a simulation) on a perfect shuffle is preferrable. D. Stott Parker in [$PARK80$] also advocates embellishing the basic interconnection network for a parallel machine when improved throughput may be obtained.

"Therefore, the architect of a new multiprocessor with a Shuffle/Exchange – type network should consider which connections will provide him with the most cost-effective switching capacity for the programs to be run on the machine, and include all of them– so the Shuffle/Exchange will comprise more connections than just a shuffle."[2]

It is still an open question whether the broadcast patterns entailed in

[2] D. Stott Parker, "Notes on Shuffle/Exchange-Type Switching Networks," *IEEE Transaction on Computers*, Vol. C-29, No. 3, March 1980, p.218.

Galil's algorithm when running on the PSP lie in the class F. If they are indeed members of this set, then the aforementioned approach merits serious consideration.

Nassimi and Sahni provide two additional solutions to the Random Access Read (RAR) data routing problem we've been considering. In this form of the problem an index $S(k)$ is contained in PR(k), $0 \leq k < N$. PE(k) is to receive data which is contained in PE($S(k)$). In our discussion $S(k)$ would point to a character in $Y^{(i)}$. Five steps are involved in the first approach: 1) SORT 2) RANK 3) CONCENTRATE 4) DISTRIBUTE and 5) GENERALIZE. These five steps are described below.

1) SORT: A *sort* will rearrange records so that they appear in non-decreasing order with respect to a given key. Let G(i) denote the record contained in PE(i) where $0 \leq i < N$ and H(i) is the key field of record G(i). It is assumed that H(i) is also present in PE(i). When a sort is complete, records are rearranged so that $H(i) \leq H(i+1), 0 \leq i < N-1$.

2) RANK: the rank of a specified record is just the number of specified records in PEs with a smaller index. Suppose we have four PEs, each containing a single record. Let the key values for these four records be $(3, 2^*, 3^*, 2^*)$. An asterisk over a key value is used to denote a flag or specified record. The ranks of the records which have been flagged are $(-, 0, 1, 2)$.

3) CONCENTRATE: Let G(i_r) where $0 \leq r \leq j < N$ be a set of records with G(i_r) initially in PE(i_r). If the records have been ranked so that $H(i_r) = r$, then a *concentrate* will result in record G(i_r) being moved to PE(r), $0 \leq r \leq j$. Assume that $G(0:3) = (A, -, -, B)$ with $i_0 = 0$ and $i_1 = 3$. Following a concentrate, $G(0:3) = (A, B, -, -)$.

4) DISTRIBUTE: Let $G(i), 0 \leq i \leq j < N$ be a set of records with $G(i)$ initially located in PE(i). $H(i)$, where $0 \leq i \leq j$, will represent a set of destinations such that $H(i) < H(i+1)$ where $0 \leq i < j$. A *distribute* will route G(i) to PE($H(i)$), $0 \leq i \leq j$. The authors comment that a distribute is merely the inverse of a concentrate. For example, if $G(0:3) = (A, B, -, -)$ and $H(0) = 0$ and $H(1) = 3$, following a distribute, $G(0:3)$ will equal $(A, -, -, B)$.

5)GENERALIZE: The purpose of a *generalize* is to make multiple copies of records. Initially record $G(i)$ is in PE(i) with $0 \leq i \leq j < N$. Each record possesses a high field, H. These H values are in increasing order, i.e. $0 \leq H(0) < H(1) < \ldots < H(j) \leq N-1$. And $H(i)$ is set equal to ∞ for $j < i < N$. Generalize will copy record $G(i)$ into PEs $H(i-1)+1$ through $H(i)$, $0 \leq i \leq j$. The authors assume that $H(-1) = 0$. Let $G(0:3) = (A, B, -, -)$ and let $H(0:3) = (2, 2, \infty, \infty)$. When the generalize operation is complete, $G(0:3) = (A, A, B, B)$.

An example illustrating how these routines may be enlisted to accomplish a RAR is provided in [NASS81a]. These routines may also be used to accomplish a RAW (Random Access Write). In a RAW instance, PE(i) will contain an index $w(i)$ which specifies a PE into which the contents of the ith PE's data register is to be written. We observe that Nassimi and Sahni's CONCENTRATE routines perform a task analagous to the hyperconcentrator in Thompson's GCN. Similarly, their DISTRIBUTE and GENERALIZE routines correspond to Thompson's infrageneralizer and their RANK and SORT procedures accomplish the permuting function performed by Thompson's connector network. The complexity for this data broadcasting approach is dominated by the SORT component of the algorithm. Sorting on an N PE PSC may be accomplished in $O(log^2 N)$ time using either of Batcher's parallel algorithm (i.e. mergesort or bitonic sort, see [$BATCH68$]). Hence, this solution to the RAR problem requires $O(\log^2 N)$ time.

One final approach is presented by Nassimi and Sahni [NASS82b]. When $N^{1+1/k}$ PEs are available, sorting may performed in $O(k \log N)$ time. This leads to an $O(k \log N)$ solution to the RAR problem. They also present a GCN which is inferior to Thompson's in terms of required contact pairs ($O(kN^{1+1/k} \log N)$ vs. $O(N \log N)$) but superior in terms of set up time ($O(k \log N)$ vs. $O(\log^4 N)$). For short records (recall that in our string matching problem, a record consists simply of a single character) it is un-

likely that any benefits will accrue from including this GCN as part of the interconnection network for the PSP. However, if we heed Galil's advice to include $\log n$ or even $\log^2 n$ characters per PE, then including Nassimi and Sahni's GCN as part of the PSP may indeed prove worthwhile. Several alternative solutions to the data broadcasting problem have been presented. It is not possible at present to gauge which strategy is optimal. Certainly further study is warranted.

Conclusion

We have outlined a parallel architecture for Galil's string matching algorithm. The "parallel string processor" (PSP) is an SIMD computer with 2^{16} PEs and a perfect shuffle interconnection network. It is intended to run as a dedicated back end processor for a host computer. The machine is microprogrammed. During every instruction cycle the control unit will broadcast a microinstruction to each PE in the melange. Each processor is endowed with a CHECK register which serves as a mask register to disable certain PEs from participating in the execution of an instruction where appropriate. Pattern and text symbols are stored one character per processing element. Local memories are enlisted when pattern and text lengths exceed 2^{16}. These memories are organized as two stacks. In one stack subsequent pattern characters are stored and in the second resides the text suffix. A Direct Memory Access unit is employed to pipeline I/O with actual processing. And finally several solutions to the data broadcasting problem have been outlined.

Our plans, however, are not ready for delivery to the silicon foundry. We believe that simulation studies are necessary to fine tune our design. Once our design has been optimized, then if it shows merit, the next step

would be to build a scaled – down prototype, perhaps with 2^6 PEs. For a prototype is the only means of sufficiently verifying a design [*COT*84]. Several issues we cited earier concerning our design merit further study.

1) What is the optimal number of pattern and text characters to store within a PE?

2) Should the microprogramming organization be horizontal or vertical?

3) Should bus lines rather than dedicated data paths be employed?

4) What is the exact structure of the DMA unit which is to be used? As cited in [*COT*84] the design of memory and I/O systems for parallel machines lags behind that of the actual computing elements.

Designing an optimal BROADCAST facility will also require simulation studies. Several open questions remain here as well. These issues are summarized below:

1) Do the permutations entailed in routing $Y^{(i)}$ admit to efficient preprocessing to run efficiently on Thompson's GCN?

2) Are these same requisite permutations to route $Y^{(i)}$ included in the class F of permutations which may be routed on Nassimi and Sahni's self – routing Benes network?

3) Is the expenditure on a GCN to augment the perfect shuffle interconnection worthwhile?

Our machine was designed with Galil's algorithm in mind. It would be interesting to see what sort of modifications would be required to run Vishkin's algorithm. And further to gauge which of these two approaches would exhibit the better response time. Also it would prove interesting to compare these two strategies against a hardware instantiation of the Rabin – Karp algorithm.

The limit to single processor performance is now in sight. This has naturally sparked a good deal of interest in parallel computing. With hardware costs continuing to plummet and as CAD tools continue to improve, special–purpose approaches to problems will become increasingly attractive. String matching is a problem domain where real–time processing is desirable and at the same time problem instances are expanding (e.g. determining genotypes in biological applications). There has been much recent interest in special purpose processors for Artificial Intelligence (see for example the Jaunary 1987 issue of IEEE Computer). Augmenting our PSP design to support parallel string replacements as would occur in Inference systems might perhaps be a worthwhile endeavor. In any case, we believe that the PSP represents an efficient mapping of a parallel algorithm onto an architecture. Its design would appear to merit further investigation.

Index of Definitions and Notation

$Y^{(i)}$ 37

References

AHO74 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

AHO75 A.V. Aho, and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search ," *Communications of the ACM*, Vol. 18, No. 6, June 1975, pp. 333 – 340.

AKL85 Selim G. Akl, *Parallel Sorting Algorithms*, Academic Press, Inc., 1985.

BAASE78 Saara Baase, *Computer Algorithms*, Addison–Wesley, Reading, Massachusetts, 1978.

BACK79 R. C. Backhouse, *Syntax of Programming Languages Theory and Practice*, Prentice–Hall, Englewood Cliffs, New Jersey, 1979.

BAKER78 T. P. Baker, "A Technique for Extending Rapid Exact Match String Matching to Arrays of More Than One Dimension," *SIAM J. Computing*, Vol. 7, No. 4, November 1978, pp. 533 – 541.

BARN68 G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The Illiac IV Computer," *IEEE Transactions on Computers*, August 1968, pp. 746 – 757.

BATCH68 K. E. Batcher, "Sorting networks and their applications," *The Proceeding of AFIPS 1968 SJCC*, 1968, pp. 307 – 314.

BATCH80 K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, C – 29, September 1980, pp. 836 – 840.

BELL71 G. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw–Hill, New York, 1971.

BOY77 R. S. Boyer, J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, Vol. 20, No. 10, October 1977, pp. 762 – 772.

COT84 Robert F. Cotellessa, editor. *Identifying Research Areas In The Computer Industry To 1995*, Noyes Publications, Park Ridge, New Jersey, 1984.

DECH86 Rina Dechter and Leonard Kleinrock, Broadcast Communications and Distributed Algorithms," *IEEE Transactions On Computers*, Vol. C – 35, No. 3, March 1986, pp. 210 – 219.

DEN78 P. J. Denning, J. B. Dennis, and J. E. Qualitz, *Machines, Languages, and Computation*, Prentice–Hall, Englewood Cliffs, New Jersey, 1978.

DOUG85 Robert J, Douglass, "Algorithms + Alchemy = Architectures," in *Algorithmically Specialized Parallel Computers*, Edited by L. Snyder, L. H. Jamieson, D. B. Gannon, and H. J. Siegel, Academic Press, Inc., 1985.

FOST80 M. J. Foster and H. T. Kung, " Design of Special Purpose VLSI Chips: Example and Opinions," Symposium on Computer Architecture, 1980, pp. 300 – 307, IEEE 1980.

GALIL79 Zvi Galil, "On Improving the Worst Case Running Time of the Boyer – Moore String Matching Algorithm," *Communications of the ACM*, Vol. 22, No. 9, September 1979, pp. 505 – 508.

GALIL83a Zvi Galil and J. Seiferas, "Time – Space – Optimal String Matching," *JCSS* 26, 1983, pp. 280 – 294.

GALIL83b Zvi Galil, "Optimal Parallel Algorithms for String Matching," Technical Report, Columbia University and Tel Aviv University, 1983.

GALIL85 Zvi Galil, "Optimal Parallel Algorithms for String Matching," *Information And Control* 67, 1985, pp. 144 – 157.

HALL80 P. A. Hall, G. R. Dowling, "Approximate String Matching," *ACM Computing Surveys*, Vol. 12, No. 4, December 1980, pp. 381 – 402.

HARR78 Michael A. Harrison, *Introduction to Formal Language Theory*, Addison–Wesley, Reading, Massachusetts, 1978.

HAYES78 J. P. Hayes, *Computer Architecture and Organization*, McGraw–Hill, New York, 1978.

HILL74 Frederick S. Hillier and Gerald J. Lieberman, *Operations Research*, 2^{nd} edition, Holden–Day, Inc., San Francisco, 1974.

HILL85 Daniel W. Hillis, *The Connection Machine* Ph.D. Dissertation, Massachusetts Institute of Technology, MIT Press, Cambridge, Massachusetts, 1985.

HILL86 Daniel W. Hillis and Guy L. Steele, Jr., "Data Parallel Algorithms," *Communications of the ACM*, Vol. 29, No. 12, December 1986, pp. 1170 – 1183.

HOP79 John E. Hopcroft, Jeffery D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison – Wesley, Reading, Massachusetts, 1979.

HOR79 R. N. Horspool, "Practical Fast Searching in Strings," *Software – Practice and Experience*, Vol. 10, 1980, pp. 501 – 506.

HWANG84 Kai Hwang and Fayé A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

JOHN68 W. L. Johnson, J. H. Porter, S. I. Ackley, D. T. Ross, "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques," *Communications of the ACM*, Vol. 11, No. 12, December 1968, pp. 805 – 813.

KARP81 Richard M. Karp and Michael O. Rabin, "Efficient Randomized Pattern – Matching Algorithms," Technical Report TR – 31 – 81, Harvard University, Aiken Computation Laboratory, Cambridge, Massachusetts, 1981.

KNUTH68 Donald E. Knuth, *The Art of Computer Programming – Fundamental Algorithms*, Vol. 1, Addison–Wesley, Reading Massachusetts, 1968.

KNUTH73 Donald E. Knuth, *The Art of Computer Programming* Vol. 3, *Sorting and Searching*, Addison–Wesley, Reading Massachusetts, 1973.

KNUTH77 Donald E. Knuth, James H. Morris, Vaughan R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal of Computing*, Vol. 6, No. 2, June 1977, pp. 323 – 350.

KRON85 Lydia Kronsjö, *Computational Complexity of Sequential and Parallel Algorithms*, John Wiley & Sons, 1985.

KRUSK83 J. B. Kruskal, "An Overview of Sequence Comparison: Time Warps, String Edits, and Macromolecules," *SIAM Review*, Vol. 25, No. 2, April 1983, pp. 201 – 237.

LANG76 T. Lang and H. Stone, "A shuffle – exchange network with simplified control," *IEEE Transactions on Computers*, Vol. C – 25, January 1976, pp. 55 – 65.

LAWR75 D. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers*, Vol. C – 24, December 1975, pp. 1145 – 1155.

LEIGH83 Frank T. Leighton, *Complexity Issues in VLSI Optimal Layouts for the Shuffle-Exchange Graph and Other Networks*, Ph.D. Dissertation, Massachusetts Institute of Technology, 1983, published by The MIT Press, Cambridge, Massachusetts, 1983.

LENF78 J. Lenfant, "Parallel Permutations of Data: A Benes network control algorithm for frequently used permutations," *IEEE Transactions on Computers*, Vol. C – 27, July 1978, pp. 637 – 647.

MYERS82 G. J. Myers, *Advances in Computer Architecture*, 2nd Edition, John Wiley and Sons, New York, 1982.

NASS81a David Nassimi and Sartaj Sahni, "Data Broadcasting in SIMD Computers," *IEEE Transactions on Computers*, Vol. C – 30, No. 2, February 1981, pp. 101 – 107.

NASS81b David Nassimi and Sartaj Sahni, "A Self – Routing Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers*, Vol. C – 30, No. 5, May 1981, pp. 332 – 340.

NASS82a David Nassimi and Sartaj Sahni, "Parallel Algorithms to Set Up the Benes Permutation Network," *IEEE Transactions on Computers*, Vol. C – 31, No. 2, Feburary 1982 , pp. 148 – 154.

NASS82b David Nassimi and Sartaj Sahni, "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network," *Journal of the Association for Computing Machinery*, Vol. 29, No. 3, July 1982 , pp. 642 – 667.

OFMAN65 J. P. Ofman, "A Universal Automaton," *Trans. Moscow Math. Soc.*, Vol. 14, 1965 (translation published by Amer. Math. Soc., Providence, Rhode Island, 1967, pp. 200 – 215).

PAD84 D.J. Paddon, Editor, *Supercomputers And Parallel Computation*, Clarendon Press, Oxford, 1984.

PARK80 D. Stott Parker, Jr., "Notes on Shuffle/Exchange Type Switching Networks," *IEEE Transactions on Computers*, Vol. C – 29, No. 3, March 1980, pp. 213 – 222.

PREP80 F. P. Preparata and J. Vuillemin, " The Cube Connected Cycles: A Versatile Network for Parallel Computation," *Communicatons of the ACM*, May 1981, pp. 300 – 309.

RABIN76 Michael O. Rabin, "Probabilisticc Algorithms," in *Algorithms and Complexity, Recent Results and New Directions*, J. F. Traub, Editor, Academic Press, New York, 1976, pp. 21 – 40.

RABIN81 Michael O. Rabin, "Fingerprinting by Random Polynomials," Technical Report, Harvard University, Aiken Computation Laboratory, Cambridge Massachusetts, 1981.

RIV77 R. L. Rivest, "On the Worst – Case Behavior of String – Searching Algorithms," SIAM Journal of Computing, Vol. 6, No. 4, December 1977, pp. 669 – 674.

SCHWA80 J.T. Schwartz, "Ultracomputers," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 4, October 1980, pp. 484 – 521.

SED83 Robert Sedgewick, *Algorithms*, Addison–Wesley, Reading Massachusetts, 1983.

SIEG79 Howard J. Siegel, "Interconnection Networks for SIMD Machines," *IEEE Computer*, Vol. 12, No. 6, June 1979, pp. 57 – 65.

SIEG85 Howard Jay Siegel, *Interconnection Networks for Large – Scale Parallel Processing*, Lexington Books, D.C. Heath and Company, Lexington, Massachusetts, 1985.

STONE71 Harold S. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers*, Vol. C –20, No. 2, February 1971, pp. 153 – 161,

TAN76 Andrew S. Tanenbaum, *Structured Computer Organization*, Prentice–Hall, Inc. Englewood Cliffs, New Jersey, 1976.

TAR79 R. E. Tarjan, A. C. Yao, "Storing a Sparse Table, " *Communications of the ACM*, Vol. 22, No. 11, November 1979, pp. 606 – 611.

THOMP78 Clark D. Thompson, "Generalized Connection Networks for Parallel Processor Intercommunication," *IEEE Transactions on Computers*, Vol. C – 27, No. 12, December 1978, pp. 1119 – 1125.

VISH85 Uzi Vishkin, "Optimal Parallel Pattern Matching in Strings," *Information And Control*, 67, 1985, pp. 91 – 113.

WAK68 Abraham Waksman, "A Permutation Network," *Journal of the Association for Computing Machinery*, Vol. 15, No. 1, January 1968, pp. 159 – 163.

YIAN83 P. N. Yianilos "A Dedicated Comparator Matches Symbol Strings Fast and Intelligently," *Electronics*, December 1983, pp. 113 – 117.