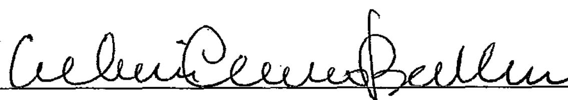
Algoritmos Paralelos para Casamento de Cadeias

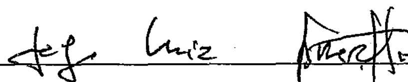
Nalvo Franco de Almeida Junior

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

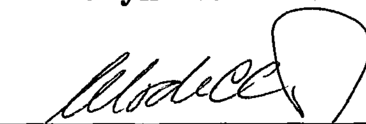
Aprovado por:



Valmir Carneiro Barbosa, Ph.D.
(Presidente)



Jayme Luiz Szwarcfiter, Ph.D.



Celso da Cruz Carneiro Ribeiro, Dr.Ing.

RIO DE JANEIRO, RJ – BRASIL

SETEMBRO DE 1992

ALMEIDA Jr., NALVO FRANCO

Algoritmos Paralelos para Casamento de Cadeias [Rio de janeiro] 1992.

VIII, 88 p. 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1992)

Tese – Universidade Federal do Rio de Janeiro, COPPE.

1 - Algoritmos Paralelos 2 - Complexidade de Algoritmos

3 - Casamento de Cadeias 4 - Casamento de Padrões

I. COPPE/UFRJ II. Título (série).

À Lígia, Nalvo (pai), Olyntha e Beth.

Agradecimentos

Aos professores Valmir e Jayme, pelas competentes orientações de tese e de programa, respectivamente.

Ao professor (e hoje colega) Sérgio Roberto de Freitas, por ter me despertado o gosto pela computação.

Ao amigo e colega Edson Norberto Cáceres, pelo apoio logístico no Rio, além das ajudas técnicas.

Aos amigos do Rio (da COPPE ou não), pelos dias de estudo e noites de chopp.

Aos colegas do DMT-UFMS que, de alguma forma, ajudaram na concretização deste trabalho.

A Deus, por eu ter tantas pessoas a quem agradecer.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

Algoritmos Paralelos para Casamento de Cadeias

Nalvo Franco de Almeida Junior

Setembro, 1992

Orientador: Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Este trabalho trata de algoritmos paralelos para resolver o problema denominado Casamento de Cadeias (*String-Matching*). Trata-se de um caso particular do problema de Casamento de Padrões (*pattern-matching*), que resume-se em procurar, num dado tipo de estrutura, um padrão, com o qual um subconjunto da estrutura se identifica.

Em particular, trata do caso em que tanto o padrão quanto a estrutura são cadeias de símbolos pertencentes a algum alfabeto.

São descritos os principais algoritmos paralelos para este problema, contidos na literatura. Descreve-se também o modelo de computação paralela para o qual todos esses algoritmos foram desenvolvidos, além dos limites inferiores para o problema.

Um dos algoritmos paralelos apresentados é modificado, no sentido de se obter a mesma complexidade num modelo mais fraco de computação paralela. Porém, o algoritmo resultante funciona apenas para alguns casos especiais, contendo restrições sobre os tamanhos do alfabeto e do padrão.

Além disso, descreve-se brevemente os mais importantes algoritmos seqüenciais para o problema.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for degree of Master of Science (M.Sc.)

Paralell Algorithms for String-Matching

Nalvo Franco de Almeida Junior

September, 1992

Thesis Supervisor: Valmir Carneiro Barbosa

Department: Computing and Systems Engineering

This thesis is concerned with the parallel algorithms that solve the string-matching problem. This problem is an example of the pattern-matching problem, which consists in searching for a pattern within a given structure. The case we are interested is the one in which both the pattern and the structure are strings of symbols that belong to a given alphabet.

In the work it is described the main parallel algorithms that solve the problem in the literature. It is also presented the parallel computing model to which all this algorithms were developed, as well as the lower bounds for the above problem.

One of the parallel algorithms presented is modified, with the objective of obtaining the same complexity in a weaker parallel computing model. However, this algorithm works only for special cases containing restrictions about the pattern and alphabet lengths.

Furthermore, it is described briefly the most important sequential algorithms for the problem.

Índice

I Introdução 1

II Principais Algoritmos Seqüenciais 7

III O Modelo PRAM 13

III.1 Introdução 13

III.2 Arquiteturas Paralelas 13

III.3 O Modelo PRAM 15

III.4 Eficiência de Algoritmos no Modelo PRAM 16

III.5 Simulações entre Modelos de PRAM 18

IV Limites Inferiores 21

IV.1 Introdução 21

IV.2 Um Limite Inferior Para Encontrar o Período 22

IV.3 Permitindo Mais Comparações a Cada Passo 27

IV.4 Algumas Considerações 28

V Principais Algoritmos Paralelos 29

V.1 Introdução 29

V.2 Um Algoritmo Ótimo para Alfabeto de Tamanho Fixo 30

V.3 Uma Nova Técnica – Os Duelos 36

V.3.1 Uma caracterização para a periodicidade 37

V.3.2 A análise do padrão 38

V.3.3 A procura do padrão pré-analisado 43

V.3.4 Complexidade 52

V.4 Uma Aplicação de Um Algoritmo de Sufixo-Prefixo 52

V.4.1 O problema de sufixo-prefixo 53

V.4.2 Um algoritmo não ótimo para a função característica 53

V.4.3 Um algoritmo ótimo para sufixo-prefixo 55

V.4.4 Um algoritmo ótimo para casamento de cadeias 59

V.5 Um Algoritmo Ótimo de Tempo $O(\log\log m)$ 60

V.5.1 Análise do Texto 60

V.5.2 A análise do padrão 61

V.6 Casamento de Cadeias com Pré-Processamento Lento 63

V.6.1 A assinatura 63

V.6.2 O pré-processamento do padrão 64

V.6.3 Duas análises não-ótimas para o texto 66

V.6.4 A análise de tempo $O(\log^* n)$ 69

V.6.5 O caso periódico 71

V.7 Um Algoritmo de Tempo Constante 71

VI Um Algoritmo Para o Modelo CREW-PRAM 78

VII Conclusões 85

Referências Bibliográficas 87

Capítulo I

Introdução

Devido ao grande número de aplicações que necessitam de máquinas rápidas, que processem muitas informações em intervalos de tempo menores, além da diminuição do custo de implementação de hardwares mais avançados, modelos computacionais não-convencionais, especificamente computadores baseados em arquiteturas paralelas, têm surgido.

A idéia de paralelismo consiste no fato de que se uma computação envolve várias operações que podem ser executadas simultaneamente, então o tempo gasto para essa computação pode ser significativamente reduzido. Portanto, nossa ferramenta computacional será um computador com várias unidades de processamento, ou **processadores**.

Podemos dizer que três aspectos envolvem a computação paralela. O primeiro diz respeito ao **modelo de computação paralela** utilizado. A definição deste modelo consiste na determinação de regras para o gerenciamento das instruções dadas aos processadores, assim como a comunicação entre eles.

Os modelos de computação paralela são classificados de acordo com seus fluxos de instruções e de dados. O modelo usado neste trabalho é o modelo pertencente à classe SIMD (*Single Instruction Stream, Multiple Data Stream*), com a comunicação entre os processadores feita através de uma memória global. Esse modelo, com esse tipo de comunicação, é também conhecido na literatura como PRAM (*Parallel Random Access Machine*). Um computador deste modelo possui N processadores idênticos, cada qual com uma memória local e uma identificação, que operam segundo um único fluxo de instruções, sobre dados possivelmente diferentes. Os processadores se comunicam através de uma memória global. Quando, por exemplo, um processador P_i deseja enviar um valor x ao processador P_j, P_i escreve x numa célula da memória global, previamente conhecida por P_j. Depois P_j lê x nesta célula. Este modelo possui um único relógio sendo, portanto, síncrono.

O segundo aspecto que envolve a computação paralela diz respeito aos sis-

temas operacionais, compiladores e linguagens de programação desenvolvidos para um determinado modelo.

O terceiro aspecto (talvez o mais importante) consiste no desenvolvimento de algoritmos para um determinado modelo de computação paralela, os quais são chamados de **algoritmos paralelos.** Esses algoritmos devem ser desenvolvidos no sentido de dividir as tarefas entre os vários processadores de forma otimizada e, a partir dos resultados alcançados por cada um deles, obter a solução final para o problema originalmente proposto.

Muitos problemas computacionais clássicos, tais como ordenação, intercalação, máximo/mínimo, computação sobre árvores, caminhos em grafos, árvores geradoras, componentes conexos são freqüentemente tratados sob uma abordagem paralela.

Neste sentido, a computação paralela, através da busca por técnicas eficientes de paralelização de algoritmos seqüenciais, tem tido extrema importância na área de desenvolvimento e análise de algoritmos em geral.

O que se procura, na verdade, é resolver um determinado problema no chamado menor **tempo paralelo** possível, sem que, para isso, seja necessário um número muito grande de processadores.

Do ponto de vista teórico, um algoritmo paralelo é considerado eficiente se pode ser executado em tempo paralelo $t = O(\log^k n)$† (chamado polilogarítmico) com um número polinimial $p(n)$ de processadores, onde n é o tamanho da entrada e k é uma constante inteira.

Um algoritmo paralelo é considerado ótimo para um determinado problema com entrada de tamanho n, se o número $p(n)$ de processadores usados por ele, multiplicado pelo tempo paralelo $t(n)$ gasto, resulta num valor $c(n)$, denominado **custo**, no máximo igual (assintóticamente) ao tempo gasto pelo melhor algoritmo seqüencial existente para resolver o problema, no pior caso.

Problemas para os quais existe algoritmo paralelo eficiente que os resolva pertencem à classe chamada NC (Nick (Pippenger)'s Class). No entanto, existem problemas inerentemente seqüenciais, de difícil paralelização. Esses problemas formam a classe dos problemas P-completos.

Este trabalho trata de algoritmos paralelos para resolver o problema denominado **casamento de cadeias** (*string-matching*), que pertence a classe NC. Trata-se de um caso particular do problema de **casamento de padrões** (*pattern-matching*), que resume-se em procurar, num dado tipo de estrutura, um padrão, com o qual um subconjunto da estrutura se identifica.

O problema de casamento de padrões tem inúmeras variantes práticas, evidentes em algumas aplicações, tais como no processamento de imagens, tratamento do código genético e em editores de texto.

Em particular, estamos interessados no caso em que: o padrão é uma cadeia

---

† Exceto nos casos mencionados, $\log x = \log_2 x$.

PAT de m símbolos; a estrutura é uma cadeia $TEXTO$ de n símbolos ($n \geq m$); e deseja-se encontrar todas as posições de $TEXTO$ onde uma ocorrência de PAT começa.

Existem variantes para esse problema que não nos interessam. Por exemplo, encontrar todas as ocorrências, mesmo com algumas falhas (também conhecido como casamento aproximado de cadeias). Ou ainda encontrar apenas a primeira ocorrência.

Trataremos aqui de algoritmos paralelos determinísticos para resolver o problema de casamento exato de cadeias.

No capítulo II descrevemos brevemente os principais algoritmos seqüencias para o problema. Em especial, os importantes algoritmos de Knuth, Morris e Pratt [KMP] e Boyer e Moore [BM]. O algoritmo de [KMP] usa a idéia de pré-processar o padrão, para tornar sua busca mais eficiente. Tal idéia é largamente usada em algoritmos posteriores, seqüenciais ou paralelos.

No capítulo III descrevemos o modelo de computação paralela PRAM, para o qual todos os algoritmos aqui descritos foram desenvolvidos. Além disso descrevemos algumas simulações entre sub-modelos de PRAM. Tais sub-modelos existem em função dos conflitos que podem surgir em virtude de leituras e principalmente escritas concorrentes (quando vários processadores tentam escrever simultaneamente na mesma célula da memória). Segundo este critério, são os seguintes os sub-modelos de PRAM: EREW-PRAM – é o sub-modelo mais fraco. Não permite nem leitura nem escrita concorente; CREW-PRAM – permite apenas leitura concorrente; CRCW-PRAM – é o sub-modelo mais forte. Permite tanto leitura quanto escrita concorrente. No caso da CRCW-PRAM, existe ainda outra classificação, no que diz respeito aos critérios estabelecidos para a resolução deste tipo de conflito. Os detalhes podem ser vistos no capítulo III.

Limites inferiores para o problema de casamento de cadeias são descritos no capítulo IV. Esses limites foram provados por Galil e Breslauer em [BG91]. Dentre eles, destaca-se o limite de tempo paralelo $O(\log\log m)$, com $O(\frac{n}{\log\log m})$ processadores de uma CRCW-PRAM.

Podemos dividir os algoritmos para casamento de cadeias em dois grupos. No primeiro estão os algoritmos que usam apenas a operação de comparação entre símbolos do alfabeto, como operação elementar. No outro estão aqueles algoritmos que, além de comparar símbolos, manipulam valores numéricos resultantes de compactações de sub-cadeias de símbolos do texto ou do padrão. Em geral, algoritmos do primeiro grupo não impõem quaisquer restrições sobre o tamanho do alfabeto de símbolos utilizado. Enquanto que o contrário acontece com os algoritmos do segundo grupo.

No capítulo V descrevemos os principais algoritmos paralelos para o problema de casamento de cadeias.

O primeiro algoritmo paralelo descrito no capítulo V é devido a Galil [G85], usa tempo paralelo $O(\log m)$ e $O(\frac{n}{\log m})$ processadores de uma CRCW-PRAM.

Este algoritmo não faz qualquer pré-processamento do padrão. No entanto funciona com essa complexidade apenas para o caso em o alfabeto é de tamanho fixo. Isso acontece porque necessita compactar $\log m$ símbolos em um único número. Usa amplamente (como a maioria dos algoritmos aqui descritos) a noção de periodicidade do padrão, mais detalhada na seção IV.2 . A cada passo i, o algoritmo testa a periodicidade do prefixo de tamanho 2^i do padrão, procurando sua ocorrência no texto.

Em seguida descrevemos o algoritmo devido a Vishkin [V85], que é ótimo, de tempo paralelo $O(\log m)$, com $O(\frac{n}{\log m})$ processadores de uma CRCW-PRAM. Este algoritmo não tem qualquer restrição sobre o tamanho do alfabeto. Possui pré-processamento e além disso usa pela primeira vez a idéia de "duelo", que consiste em, dadas duas posições muito próximas no texto, confrontá-las em tempo constante de tal forma que pelo menos uma delas possa ser descartada de uma possível ocorrência do padrão. Essa idéia, aliás, é amplamente usada nos algoritmos posteriores. Além disso, propõe uma interessante caracterização para a periodicidade do padrão, através da construção de um vetor, de tal forma que a análise do padrão, ou seu pré-processamento, se resume nessa construção. Por meio de consultas feitas neste vetor, fazem-se os duelos (cada um em tempo constante), eliminando-se assim várias posições do texto a cada passo.

O terceiro algoritmo descrito no capítulo V é um algoritmo ótimo, também de tempo paralelo $O(\log m)$ e $O(\frac{n}{\log m})$ processadores de uma CRCW-PRAM, devido a Kedem, Landau e Palem [KLP], resultante da aplicação direta de um outro algoritmo ótimo, este para o problema de sufixo-prefixo, que consiste em, dadas duas cadeias A e B, encontrar quais os prefixos de B são iguais ao sufixos de A. Tal algoritmo (para sufixo-prefixo) tem a mesma complexidade e usa a idéia de dividir os sufixos de A e os prefixos de B em classes de equivalência, de tal forma que a descoberta da classe de equivalência de uma cadeia depende das classes às quais pertencem as suas sub-cadeias menores.

A seguir, ainda no capítulo V, descrevemos o algoritmo ótimo de tempo paralelo $O(\log\log m)$, que usa $O(\frac{n}{\log\log m})$ processadores de uma CRCW-PRAM, devido a Breslauer e Galil [BG90]. É baseado quase que inteiramente nos algoritmos de [V85] e [G85]. Ganha na complexidade de tempo paralelo em função da descoberta de que o duelo, inicialmente proposto em [V85], funciona da mesma forma que o máximo. Em outras palavras, encontrar o máximo entre k elementos, segundo [SHI81], leva tempo paralelo $O(\log\log k)$, com $O(\frac{k}{\log\log k})$ processadores de uma CRCW-PRAM. Então pode-se duelar k posições do texto com a mesma complexidade.

No capítulo V, descrevemos também um algoritmo ótimo, de tempo paralelo $O(\log^* n)$ †, e $O(\frac{n}{\log^* n})$ processadores. Esse algoritmo, devido a Vishkin [V90], usa um pré-processamento bastante lento, mais precisamente de tempo paralelo $O(\frac{\log^2 m}{\log\log m})$, não se enquadrando, portanto, nos limites inferiores descritos

† $\log^* x = \min_i\{\log^{(i)} x \leq 2\}$, onde $\log^{(1)} x = \log x$ e $\log^{(i)} x = \log\log^{(i-1)} x$.

no capítulo IV, já que lá estão apenas os algoritmos que incluem, em seus custos, qualquer tipo de pré-processamento. Esse algoritmo propõe a idéia da "assinatura", um conjunto (de tamanho menor que $\log m$) de posições do padrão tal que são testadas apenas essas posições inicialmente, a cada posição candidata a uma ocorrência do padrão no texto.

Para finalizar o capítulo V, descrevemos um algoritmo ótimo, de tempo paralelo constante, devido a Galil [G92], que pelo mesmo motivo do algoritmo anterior, não se enquadra nos limites descritos no capítulo IV. A idéia também é de escolher algumas posições específicas do padrão, as quais serão inicialmente utilizadas para os testes de emparelhamento com posições do texto.

A parte chamada **análise do texto** (busca do padrão já pré-analisado) do algoritmo de Vishkin [V85] pode ser executada em tempo paralelo $O(\log m)$, com $O(\frac{n}{\log m})$ processadores de uma CREW-PRAM, portanto sem escritas concorrentes. Porém, isto não acontece com sua análise do padrão, que é executada em tempo $O(\log^2 m)$ com $O(\frac{n}{\log^2 m})$ processadores nesse modelo. Aliás, os melhores algoritmos de casamento de cadeias desenvolvidos para o modelo CREW-PRAM são aqueles feitos para o modelo CRCW-PRAM , com uma perda logarítimica na eficiência.

Nós propomos, no capítulo VI, um algoritmo ótimo de tempo paralelo $O(\log m)$, com $O(\frac{n}{\log m})$ processadores de uma CREW-PRAM, para o pré-processamento do padrão, mais precisamente para a construção do vetor utilizado no pré-processamento do algoritmo de [V85]. Porém, nosso algoritmo funciona com esta complexidade apenas nos casos em que $m = O(\log^2 n)$ e o alfabeto é de tamanho fixo.

Na verdade, inicialmente propomos um algoritmo para o caso geral. Só que este algoritmo pertence ao grupo dos algoritmos que manipulam valores numéricos resultantes da compactação de cadeias de símbolos do padrão. Além disso, manipula valores numéricos muito grandes, mais precisamente de tamanho $O(n^m)$. Assim, o custo de uma operação de leitura/escrita na memória da PRAM passa a ser $O(m)$ (ao invés de $O(1)$). Com a imposição das restrições quanto ao tamanho do alfabeto e o tamanho do padrão, obtemos o algoritmo ótimo citado no parágrafo anterior.

A tabela I.1 a seguir resume, em termos de complexidade, os algoritmos descritos no capítulo V, executados no modelo CRCW-PRAM. Dentre todos, o único que necessita de alguma restrição quanto ao fato do alfabeto ser de tamanho fixo é o de [G85].

Algoritmo	Análise do padrão	Tempo de busca	Tempo total	Número de proc.
[G85]	—	$O(\log m)$	$O(\log m)$	$O(\frac{n}{\log m})$
[V85]	$O(\log m)$	$O(\log m)$	$O(\log m)$	$O(\frac{n}{\log m})$
[KLP]	—	$O(\log m)$	$O(\log m)$	$O(\frac{n}{\log m})$
[BG90]	$O(\log\log m)$	$O(\log\log m)$	$O(\log\log m)$	$O(\frac{n}{\log\log m})$
[V90]	$O(\frac{\log^2 m}{\log\log m})$	$O(\log^* n)$	$O(\frac{\log^2 m}{\log\log m})$	$O(\frac{n}{\log^* n})$
[G92]	$O(\frac{\log^2 m}{\log\log m})$	$O(1)$	$O(\frac{\log^2 m}{\log\log m})$	$O(n)$

Tabela I.1

A tabela I.2 a seguir resume as complexidades dos algoritmos descritos no capítulo V, além do nosso, proposto e descrito no capítulo 6, quando executados no modelo CREW-PRAM.

Algoritmo	Tempo total	Número de processadores
[G85]	$O(\log^2 m)$	$O(\frac{n}{\log^2 m})$
[V85]	$O(\log^2 m)$	$O(\frac{n}{\log^2 m})$
[KLP]	$O(\log^2 m)$	$O(\frac{n}{\log^2 m})$
[BG90]	$O(\log m \cdot \log\log m)$	$O(\frac{n}{\log\log m})$
[V90]	$O(\log m \cdot \frac{\log^2 m}{\log\log m})$	$O(\frac{n}{\log m \cdot \log^* n})$
[G92]	$O(\log m \cdot \frac{\log^2 m}{\log\log m})$	$O(n)$
Este trabalho	$O(\log m)$	$O(\frac{n}{\log m})$

Tabela I.2

Como podemos ver na tabela I.2, o nosso algoritmo, apesar das restrições do alfabeto (fixo) e do tamanho do padrão ($m = O(\log^2 m)$), tem complexidade de tempo paralelo melhor do que a de [V85], que não tem qualquer restrição, e melhor que a de [G85], que possui apenas a restrição quanto ao tamanho do alfabeto.

Finalmente, no capítulo VII tecemos algumas considerações finais a respeito do trabalho, em especial a respeito dos algoritmos paralelos apresentados nos capítulos V e VI.

Capítulo II

Principais Algoritmos Seqüenciais

Descreveremos brevemente neste capítulo os principais algoritmos seqüenciais para o problema de casamento de cadeias.

O algoritmo mais óbvio é o chamado "algoritmo de força-bruta". Consiste em, para cada possível posição do texto $T[1 \ldots n]$, na qual o padrão $PAT[1 \ldots m]$ pode ocorrer, testar, símbolo por símbolo, se de fato, o padrão ocorre naquela posição.

O número de comparações feito pelo algoritmo de força-bruta se aproxima do número de símbolos do texto à medida em que cresce o tamanho do alfabeto, já que as "falsas" ocorrências tendem a ser detectadas já entre os símbolos iniciais do padrão.

Por outro lado, no caso geral, em que o alfabeto não é muito grande, envolvendo portanto a maioria das aplicações, este algoritmo é ruim. No pior caso, por exemplo, o número de comparações é igual a $m \cdot (n - m + 1)$. Tal número é, portanto, $O(m \cdot n) = O(n^2)$, já que $m \leq n$ (em geral m é muito pequeno comparado a n).

A complexidade do algoritmo de força bruta está diretamente ligada ao fato de que, ao se descobrir que $T[i + p] \neq PAT[1 + p]$, $0 \leq p \leq m - 1$, tem-se que "voltar" $p - 1$ posições no texto, para começar os próximos testes no seu $(i + 1)$-ésimo símbolo, $1 \leq i \leq n - m$.

Essa preocupação em não voltar no texto, dada a descoberta de uma não ocorrência do padrão numa determinada posição, levou Knuth, Morris e Pratt a desenvolverem um algoritmo, descrito em [KMP], o qual chamaremos de *algoritmo de KMP*, que encontra todas as ocorrências de um padrão em tempo $O(m + n)$.

Tal algoritmo baseia-se fundamentalmente na idéia de que, quando uma falha é detectada na $(i + p)$-ésima posição do texto, ou seja, quando descobre-se que $PAT[1 \ldots p] = T[i \ldots i{+}p{-}1]$ e $PAT[1{+}p] \neq T[i{+}p]$, $T[i \ldots i{+}p]$ é conhecido.

A idéia é então tirar proveito dessa informação ao invés de voltar para a $(i+1)$-ésima posição do texto, da seguinte forma. Digamos que uma falha ocorreu na $(i+p)$-ésima posição do texto. Então, caso se saiba, *a priori*, o tamanho do maior prefixo $PAT[1 \ldots t]$ de PAT, $t \leq p$, tal que

$$PAT[1 \ldots t] = T[i+p+1-t \ldots i+p],$$

pode-se continuar os testes a partir da posição t de PAT e $i+p$ de T.

Para se descobrir tal t é preciso que haja um pré-processamento do padrão. Esse pré-processamento consiste na construção de um vetor de m posições que indica, para cada j, $1 \leq j \leq m$, quantas posições voltar, no padrão, para o próximo teste. Em outras palavras, o valor armazenado na posição j do vetor diz qual é a posição de PAT que deveria estar sendo testada quando se encontrou uma falha na j-ésima posição do padrão. Na realidade diz, de uma forma indireta, quantas posições o padrão deve ser "empurrado" para a direita, sobre o texto.

O pré-processamento pode ser feito facilmente usando emparelhamentos do padrão sobre ele mesmo e custa tempo $O(m)$.

O algoritmo de KMP funciona então da seguinte forma. A cada passo do processo de busca, move-se ou o apontador i do texto, ou o apontador j do padrão, ou ambos, e cada um desses se move no máximo n vezes. Logo no máximo $2 \cdot n$ passos são executados, desde que o vetor de pré-processamento esteja pronto. Os dois apontadores se movem juntos à medida em que não são encontradas falhas. Ou seja, no caso em que $T[i] = PAT[j]$. Quando uma falha é encontrada, digamos na posição i do texto, o apontador j recebe o valor r armazenado na j-ésima posição do vetor de pré-processamento. É como se o padrão fosse "empurrado" para a direita $j-r$ posições sobre o texto.

O custo total do algoritmo de KMP é $O(m+n)$, já incluindo o custo de tempo do pré-processamento.

Os autores de [KMP] observam que, como num editor de textos os padrões são geralmente pequenos, é mais eficiente construir uma espécie de autômato de reconhecimento do padrão a ser procurado, como pré-processamento. O algoritmo de busca seria então a execução deste autômato, tendo o texto como entrada.

Tal máquina consiste de *estados* e *transições*. A partir de cada estado há duas transições possíveis: uma *transição de emparelhamento* e uma *transição de falha*. Cada estado contém instruções a serem executadas, que são instruções de **goto**. Quando a máquina se encontra num estado α, executa somente uma instrução: "se o símbolo atual (do texto) é igual a α então passe para o próximo símbolo do texto e vá para a transição de emparelhamento, caso contrário vá para a transição de falha". O primeiro estado sempre vai para a transição de emparelhamento e o último é o estado de parada. A máquina só se encontra nele no caso do padrão ter sido reconhecido totalmente no texto.

O algoritmo de KMP descrito acima encontra somente a primeira ocorrência do padrão no texto. A generalização para o nosso problema é facilmente obtida

fazendo-se com que um símbolo não pertencente ao alfabeto seja concatenado ao final do padrão. Quando tal símbolo é comparado a algum outro do texto, significa que o padrão original foi encontrado. Prossegue-se, então, normalmente a busca por outras ocorrências. Essa generalização não aumenta a complexidade do algoritmo.

A principal característica do algoritmo de KMP, além de ter sido um dos primeiros algoritmos seqüenciais ótimos para o problema de casamento de cadeias, é o de fazer uso das chamadas *failure functions*, que nada mais são do que funções que dizem o que fazer quando ocorre uma falha na busca. No caso deste algoritmo tais funções se traduzem no vetor de pré-processamento.

Unindo os conceitos de *failure functions* com os de autômatos, Alfred Aho e Margaret Corasick descreveram, em [ACO], um algoritmo que encontra todas as ocorrências de qualquer padrão pertencente a um conjunto finito $K = \{y_1, \ldots, y_k\}$ de padrões num texto.

A primeira etapa do algoritmo consiste na construção de uma máquina que reconhece todos os padrões pertencentes a K. Essa máquina nada mais é do que a tradução do comportamento de três funções: uma função g, que faz uma transição de emparelhamento de uma estado para o outro da máquina, ou seja, significa que o símbolo corrente do texto é igual ao símbolo corrente do padrão que está sendo, até então, reconhecido; uma função de falha (*failure function*) f, que é uma função de transição de falha, ou seja, diz qual o símbolo de qual padrão de K deveria estar sendo testado naquela posição do texto onde foi detectada a falha (de certa forma equivalente à transição de falha do algoritmo de KMP); e finalmente uma função de saída s que fornece, a cada transição para um estado α, o conjunto $s(\alpha)$ dos padrões de K encontrados quando da leitura do símbolo corrente do texto.

O conjunto $s(\alpha)$ pode ser vazio (no caso de nenhum padrão de K ter sido reconhecido quando da leitura do símbolo corrente do texto e a conseqüente transição para o estado α), assim como ter mais de um elemento, podendo, portanto, os padrões de K serem sobrepostos entre si.

A construção de tal máquina é feita em duas etapas. Na primeira são determinados os estados e a função g é construída. Na segunda etapa é construída a função de falhas f. A computação da função s é iniciada na primeira etapa e concluída na segunda.

Para a construção da função g utiliza-se um grafo, inicialmente com apenas um vértice, que representa o estado inicial da máquina. A partir de cada padrão lido é adicionado ao grafo um caminho orientado, com início no vértice inicial. Novos vértices e arestas são adicionados à medida em que são lidos os símbolos de um determinado padrão de K. Quando um padrão é lido por inteiro, ele é adicionado ao conjunto $s(\alpha)$, onde α é o estado final representado pelo último vértice no caminho orientado originado por esse padrão . Novas arestas são adicionadas ao grafo somente quando necessário, caracterizando-se assim o determinismo da máquina. Ao final da construção o grafo obtido representa a função g.

A função f é construída a partir da função g, de tal forma que um vértice

v (um estado) com distância $d(v)$ do vértice inicial é tal que $f(v)$ depende dos vértices w tais que $d(w) < d(v)$. Isto faz com que o cálculo de f seja feito a partir do vértice inicial. Durante a computação de f, alguns valores da função s são alterados.

O custo de tempo da construção da máquina é linearmente proporcional à soma dos tamanhos dos padrões pertencentes a K.

Depois de construída a máquina que reconhece os padrões de K, vem a segunda parte do algoritmo, que consiste na aplicação do texto como entrada para a máquina.

Um ciclo de funcionamento da máquina é definido da seguinte forma. Seja e o estado corrente e a o símbolo corrente do texto:

(1) Se $g(e,a) = e'$ então a máquina faz uma transição de emparelhamento, ou seja, passa para o estado e' e lê o próximo símbolo do texto. Além disso, se $s(e') \neq \emptyset$, então emite o conjunto $s(e')$ e a posição corrente do texto.

(2) Se $g(e,a)$ produz uma falha, então a máquina consulta a função f e faz uma transição de falha. Se $f(e) = e'$, repete o ciclo tendo e' como estado corrente e a como símbolo corrente.

No início, o estado corrente é o inicial e o primeiro símbolo do texto é o símbolo corrente. A máquina então processa o texto fazendo um ciclo a cada símbolo do texto.

Usando as funções g, f e s, o algoritmo descrito executa menos do que $2 \cdot n$ transições de estado no processamento de um texto de tamanho n.

No caso particular em que $\mid K \mid = 1$, ou seja, para o nosso problema específico, o algoritmo de Aho e Corasick é ótimo, já que usa tempo $O(m)$ para construir a máquina de reconhecimento e tempo $O(n)$ para executá-la no texto. Usando, portanto, um tempo total de $O(m+n)$.

Robert Boyer e Strother Moore, em [BM], descreveram um algoritmo seqüencial para casamento de cadeias, baseado na seguinte idéia. Digamos que se queira saber se $PAT[1\dots m]$ ocorre na i-ésima posição de $T[1\dots n]$ e $\beta = T[i+m-1]$ não ocorra em PAT. Daí, com certeza, PAT não ocorre em nenhuma das posições $i, i+1, \dots, i+m-1$ do texto. Portanto, se o teste entre β e $PAT[m]$ é feito logo que se queira saber se PAT ocorre na i-ésima posição do texto, podemos, caso $\beta \neq PAT[j]$, $1 \leq j \leq m$, eliminar m posições do texto. Ou seja, o padrão pode ser "empurrado" m posições para a direita, no texto.

A generalização dessa idéia, que consiste em testar os símbolos da direita para a esquerda, envolve também os casos em que β ocorre em PAT.

Tem-se, portanto, alguns casos a serem considerados. O primeiro é aquele em que a última ocorrência de β em PAT está a $delta_1$ posições do final de PAT. Ou seja, $delta_1 = m-p$, onde p é a posição onde β ocorre pela última vez em PAT. Se β não ocorre em PAT, tem-se $delta_1 = m$. Em qualquer destes casos pode-se

empurrar PAT $delta_1$ posições para a direita, sobre o texto. O segundo caso é aquele em que $\beta = PAT[m]$. Testa-se então o símbolo anterior a β no texto com $PAT[m-1]$ e assim sucessivamente, até encontrar um emparelhamento total do padrão no texto, ou encontrar um novo símbolo β no texto, a r posições do final de PAT, que falha no teste. Pode-se aqui aplicar novamente o primeiro caso para o novo símbolo β, através do $delta_1$ deste novo símbolo, ou descobrir qual a segunda ocorrência, da direita para a esquerda, do padrão $subPAT = PAT[m-r+1 \ldots m]$. Se esta segunda ocorrência, da direita para a esquerda, de $subPAT$ estiver a $delta_2$ posições da primeira, então pode-se mover o padrão de $delta_2$ posições para a direita, sobre o texto.

Note que $delta_1$ deve ser calculado para cada símbolo do texto, ou seja, para cada símbolo do alfabeto. E $delta_2$ para cada posição do padrão. Ou seja, o algoritmo de Boyer e Moore necessita de pré-processamento, aqui traduzido pelas tabelas $delta_1$ e $delta_2$.

A construção de $delta_1$ requer tempo $O(|\Sigma| + m)$, onde Σ é o alfabeto. Enquanto que a construção de $delta_2$ requer tempo $O(m)$.

Depois de prontas as tabelas $delta_1$ e $delta_2$, o algoritmo de Boyer e Moore funciona da seguinte maneira. Suponha que a última posição do padrão seja testada com a i-ésima posição do texto. Testa-se, a partir daí, da direita para a esquerda, até que se encontre o emparelhamento de todo o padrão, ou até que se encontre uma falha. Ao se encontrar uma falha, "empurra-se" o padrão $\max\{delta_1[T[i-r]], delta_2[r]\}$, onde $r = m - j$, tal que a falha ocorreu na j-ésima posição de PAT.

O tempo gasto na execução do algoritmo é $O(m+n)$.

Recentemente, em [MCDP], Crochemore e Perrin descreveram um algoritmo que pode ser considerado como intermediário entre o algoritmo de KMP e o de Boyer e Moore. Ele executa os testes tanto da esquerda para a direita, como ao contrário. Os testes começam, em geral, numa posição intermediária do padrão. Posição essa encontrada com base no que é chamado de "Teorema de Fatorização Crítica", que permite a determinação eficiente de posições críticas no padrão, as quais são interessantes para a escolha de tal posição intermediária.

O algoritmo tem complexidade $O(m+n)$, como o de KMP e o de Boyer e Moore. Por outro lado, usa espaço de memória de tamanho constante. O que não acontece com os outros dois algoritmos, que usam espaço de tamanho $O(m)$.

O algoritmo que acabamos de citar mantém a preocupação de economia de espaço, aparente também no trabalho apresentado por Galil e Seiferas, em [GS], o qual também utiliza espaço de memória de tamanho constante. Este algoritmo, aliás, melhorou o resultado apresentado pelos mesmos autores em [GS0], em que a quantidade de memória era de $O(\log m)$.

O algoritmo de Galil e Seiferas, de [GS], se baseia também num pré-processamento do padrão , onde este é decomposto de tal maneira que se tem, para cada falha encontrada em determinada posição, quanto que o padrão deve ser

"empurrado" para a direita no texto. Esse valor depende apenas da periodicidade do padrão. Logo, esse algoritmo elimina a necessidade das *failure functions*, introduzidas inicialmente no trabalho de Knuth, Morris e Pratt. Sua complexidade de tempo é também $O(m + n)$. Muitos algoritmos paralelos, como veremos, se baseiam em propriedades da periodicidade do padrão.

Antes de encerrarmos este capítulo, vale a pena tecer alguns comentários a respeito do trabalho feito recentemente por Colussi, Galil e Giancarlo, descrito em [CGG], onde são estabelecidos limites inferiores e superiores para o problema de casamento de cadeias.

Seja $c(n, m)$ o número máximo de comparações feito por um algoritmo de casamento de cadeias, tendo como entradas um texto de tamanho n e um padrão de tamanho m. É sabido que $n \leq c(n, m) \leq 2n - m + 1$. O limite inferior é um limite natural do problema, enquanto que o superior é devido ao algoritmo de KMP. Esses limites foram melhorados em [CGG].

O novo limite superior obtido foi $c(n, m) \leq n + \lfloor \frac{n-m}{2} \rfloor$, através de uma técnica baseada em provas de corretude de programas. Escreve-se uma prova de corretude do algoritmo de força bruta. Depois descobrem-se partes nas quais o algoritmo não usou algumas informações. Usa-se então tais informações e deriva-se o algoritmo de KMP. Aplica-se, daí, a mesma técnica para o algoritmo de KMP, obtendo-se assim um algoritmo melhor. Este algoritmo tem a complexidade do novo limite, além de ser, segundo os autores, bem mais simples que o de KMP. Baseia-se em selecionar algumas posições do padrão e testá-las, da esquerda para a direita. As restantes são testadas da direita para a esquerda, caso as posições selecionadas não tenham gerado falhas.

Os limites inferiores foram os seguintes, para um alfabeto de pelo menos dois símbolos: $c(n, m) \geq n$, para $0 < m \leq 2$ e $\forall n$; e ainda $c(n, m) \geq n + \lfloor \frac{n}{2m} \rfloor$.

Capítulo III

O Modelo PRAM

III.1 Introdução

Os algoritmos paralelos de casamento de cadeias que estudaremos foram desenvolvidos para um determinado modelo de Computação Paralela. Neste capítulo vamos descrever este modelo.

Na seção III.2 fazemos uma breve descrição de arquiteturas paralelas. Na seção III.3 descrevemos detalhadamente o modelo de computação paralela denominado PRAM, além de classificar seus sub-modelos segundo critérios de concorrência. Na seção seguinte apresentamos medidas de eficiência de algoritmos em tal modelo. Finalmente, na seção III.5, fazemos algumas simulações entre os vários modelos de PRAM.

III.2 Arquiteturas Paralelas

O interesse na Computção Paralela vem tendo considerável crescimento devido, principalmente, ao rápido declínio do custo de implementação de máquinas com vários processadores.

Um grande número de modelos de computação paralela tem surgido até hoje. Esses modelos se diferenciam pela forma com que são arranjados, física e logicamente, seus componentes (processadores); pelo tipo de comunicação definida

entre eles; e pelo modo de se definir os fluxos de dados e de instruções [AKL]. Enfim, pela sua arquitetura.

Em função dessas diferenças, podemos classificar os modelos de computação paralela de acordo com as seguintes arquiteturas:

(i) *Single Instruction Stream, Single Data Stream* (SISD);

(ii) *Multiple Instruction Stream, Single Data Stream* (MISD);

(iii) *Single Instruction Stream, Multiple Data Stream* (SIMD);

(iv) *Multiple Instruction Stream, Multiple Data Stream* (MIMD);

Um computador de arquitetura SISD contém um único processador, que recebe da unidade de controle um único fluxo de instruções, que são executadas sobre um único fluxo de dados. Um algoritmo escrito para uma máquina pertencente a esta classe é chamado de seqüencial ou serial.

Na arquitetura MISD um computador tem uma unidade de controle para cada um de seus processadores, cada par destes tem seu próprio fluxo de instruções. Diferentes instruções são executadas sobre o mesmo conjunto de dados, já que existe apenas um fluxo de dados.

Uma máquina pertencente à arquitetura MIMD consiste de N unidades de controle, cada qual com seu fluxo de instruções. Cada um dos N processadores tem seu próprio fluxo de dados. Algoritmos escritos para máquinas desta arquitetura usam conceitos como: processo, tarefa, entre outros.

O modelo de computação paralela que usamos neste trabalho se encaixa na arquitetura SIMD, que consiste de máquinas que contêm uma única unidade de controle, que emite instruções para seus N processadores, que por sua vez recebem dados de uma única memória, através de N fluxos exclusivos de dados. Veja a Figura III.1 .

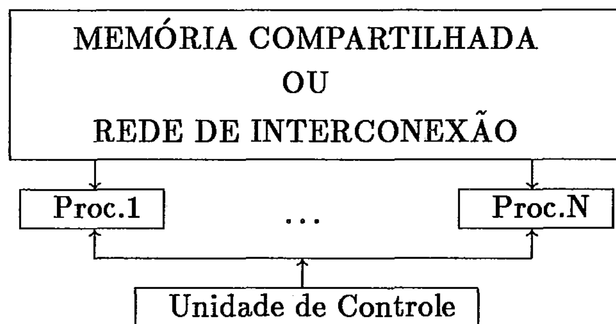


Figura III.1

III.3 O Modelo PRAM

Como dissemos anteriormente, o modelo que usaremos se enquadra na arquitetura SIMD, com a particularidade de ter, como meio de comunicação entre os processadores, uma memória global, compartilhada, ao invés de uma rede de interconexão.

Este modelo é denominado *Parallel Random Access Machine* (PRAM). Aqui N processadores, cada qual com seu identificador próprio, usam uma memória comum da mesma forma que N pessoas usam um quadro de avisos para se comunicarem. Ou seja, quando dois processadores desejam se comunicar, o fazem por intermédio da memória. Logo, quando um processador P_i deseja enviar um número ao processador P_j, deve escrevê-lo numa célula da memória, cujo endereço é conhecido por P_j. P_j então lê, nesta célula, o valor escrito por P_i. Além disso, cada processador tem uma memória local, própria.

Uma máquina PRAM, então, consiste de N máquinas RAM [AHU] acrescidas das instruções paralelas e da memória global. Assim, a unidade de controle contém as instruções da máquina, executa-as em série e as distribui em paralelo para os processadores. Um processador paralelo está ativo somente quando uma instrução é a ele atribuída. Dois processadores ativos ao mesmo tempo devem executar a mesma instrução , mas podem operar sobre dados distintos. Caracteriza-se, assim, o sincronismo do modelo.

Em uma unidade de tempo, um processador pode ler numa célula da memória global, ou escrever nela, ou ainda escrever ou ler numa célula de sua memória local.

Uma importante questão relacionada ao modelo PRAM diz respeito ao que acontece quando mais de um processador tenta, ao mesmo tempo, acessar a mesma célula da memória global.

Em função desta preocupação, as PRAM's são classificadas da seguinte forma:

(i) *Exclusive Read, Exclusive Write* (EREW-PRAM) – Tal máquina não permite nem leitura, nem escrita concorrente. É o modelo mais fraco de PRAM. O comportamento da máquina fica indefinido quando se tenta violar esta regra;

(ii) *Concurrent Read, Exclusive Write* (CREW-PRAM) – Este modelo permite somente leitura concorrente. De novo, quando há uma tentativa de escrita concorrente, a máquina tem seu comportamento indefinido;

(iii) *Concurrent Read, Concurrent Write* (CRCW-PRAM) – Este modelo permite tanto leitura, como escrita concorrente.

Para o modelo CRCW-PRAM há o problema de se decidir qual o critério a ser usado quando dois ou mais processadores tentam escrever concorrentemente numa mesma célula da memória global. A preocupação em resolver conflitos desta natureza fez com que surgissem alguns modelos de CRCW-PRAM [EPG], quais sejam:

(a) CRCW-PRAM-Fraca – Neste modelo todos os processadores envolvidos na escrita concorrente são obrigados a escrever o valor nulo (ou o valor 1, em alguns casos);

(b) CRCW-PRAM-Modo-comum – Não há restrições quanto ao valor a ser escrito pelos processadores. No entanto, este deve ser igual para todos eles;

(c) CRCW-PRAM-Arbitrária – Em tal modelo de CRCW-PRAM os processadores que farão a escrita concorrente podem escrever quaisquer valores. Prevalece o valor escrito por qualquer um deles, arbitrariamente. Logo, a mesma instrução , executada repetidamente com os mesmos dados, pode causar resultados diferentes;

(d) CRCW-PRAM-Prioridade – Neste modelo vence o processador que tiver o maior identificador (ou menor, equivalentemente);

(e) CRCW-PRAM-Forte – Aqui prevalece o valor escrito pelo processador que estiver escrevendo o maior (ou o menor) valor.

Na seção III.5 faremos algumas simulações entre os modelos de PRAM descritos acima. Antes, porém, veremos rapidamente os conceitos envolvendo a eficiência de algoritmos paralelos desenvolvidos para o modelo PRAM.

III.4 Eficiência de Algoritmos no Modelo PRAM

A eficiência dos algoritmos escritos para os vários modelos de PRAM pode ser medida levando-se em conta três parâmetros, todos em função do tamanho da entrada do problema: o tempo paralelo do algoritmo, que consiste no tempo total gasto por ele, com a restrição de cada computação efetuada em paralelo contribui com uma unidade para o tempo total; o número de processadores envolvidos no algoritmo e o espaço utilizado por ele (este último parâmetro é pouco usado).

Em geral, a eficiência de um algoritmo paralelo é dada pelo tempo paralelo e pelo número de processadores. Por exemplo: "O algoritmo A gasta tempo $O(\log n)$ usando $O(n)$ processadores de uma CRCW-PRAM-Forte".

Outras medidas de eficiência importantes são: **aceleração** e o **custo**, definidas abaixo.

Sejam A o melhor algoritmo seqüencial conhecido para resolver um problema, B um algoritmo paralelo para resolver o mesmo problema, $T(A)$ o tempo seqüencial gasto por A e $T_P(B)$ o tempo paralelo gasto por B, tendo a entrada do mesmo tamanho que a de A. Então

$$\text{Aceleracão}(B) = \frac{T(A)}{T_P(B)}.$$

Note que, quanto maior a aceleração de um algoritmo, melhor ele é. O pior caso é aquele em que o aceleração é igual a 1. Pois, se for menor que 1, o algoritmo paralelo se torna inviável.

O custo de um algoritmo paralelo nada mais é do que o produto do tempo paralelo gasto, pelo número de processadores usados. O custo é sempre maior ou igual ao tempo do melhor algoritmo seqüencial para o problema. Pois, caso contrário, haveria um algoritmo seqüencial melhor do que o ótimo. Logo, um algoritmo paralelo é ótimo quando seu custo for igual ao tempo do melhor algoritmo seqüencial para o problema.

Podemos relacionar a aceleração do algoritmo B com o número $N(B)$ de processadores usados por ele da seguinte forma:

$$T_P(B) \cdot N(B) \geq T(A).$$

Logo,

$$N(B) \geq \frac{T(A)}{T_P(B)} = \text{aceleracão}(B).$$

Assim, a aceleração de B é no máximo igual ao número de processadores usados pelo algoritmo B.

Um algoritmo paralelo é considerado eficiente quando seu tempo paralelo é expresso por uma função polinomial no logaritmo do tamanho da entrada (tal função é chamada de *polilog*), usando um número polinomial, no tamanho da entrada, de processadores.

A classe dos problemas que admitem algoritmos paralelos eficientes é chamada de NC.

Como veremos posteriormente, nosso problema de casamento de cadeias pertence à classe NC.

III.5 Simulações entre Modelos de PRAM

Um algoritmo desenvolvido para um modelo fraco de PRAM certamente pode ser usado num modelo mais forte, sem que haja perda, tanto no tempo, quanto no número de processadores usados. Por outro lado, gostaríamos de usar qualquer algoritmo sobre qualquer modelo de PRAM. Logo é importante sabermos simular eficientemente algoritmos desenvolvidos para modelos mais fortes em modelos mais fracos de PRAM.

Antes de fazermos algumas simulações, vamos ver um importante resultado, conhecido como Teorema de Brent, que se constitui numa simulação entre duas PRAM's de mesmo modelo. Na verdade consiste , dado um algoritmo desenvolvido para um determinado modelo de PRAM, na tentativa de se diminuir o número de processadores usados por ele, sem perda na complexidade de tempo paralelo. Tal resultado será amplamente usado por nós posteriormente.

Teorema III.1 – Suponha que um problema possa ser resolvido através de um algoritmo paralelo utilizando tempo t e m operações . Então este algoritmo pode ser implementado em tempo $O(\frac{m}{p} + t)$, utilizando p processadores.

Prova – Suponha que no passo paralelo i, $1 \leq i \leq t$, o algoritmo tenha computado m_i operações . Logo $m = \sum_{i=1}^{t} m_i$. Se usarmos p processadores para executarmos o passo i, o tempo total consumido neste passo será $O(\frac{m_i}{p} + 1)$. Ou seja, o tempo paralelo gasto pelo algoritmo, em sua totalidade, é $O(\sum_{i=1}^{t}(\frac{m_i}{p}+1)) = O(\frac{m}{p}+t)$. ∎

Como dissemos anteriormente, um algoritmo desenvolvido para um modelo fraco de PRAM pode ser simulado num modelo mais forte, sem perdas no tempo paralelo e na quantidade de processadores usados.

Como exemplo, vejamos a simulação de uma escrita concorrente, feita inicialmente para o modelo CRCW-PRAM-Prioridade para o modelo CRCW-PRAM-Forte. Antes da escrita simultânea, os processadores escrevem (simultaneamente), num mesmo endereço x da memória global, o seu identificador. Como resultado, o conteúdo de x será o valor y do processador com maior identificador. Em seguida, apenas o processador y escreve o valor desejado originalmente. A simulação usa apenas duas, portanto um número constante de unidades de tempo.

A seguir, veremos outro resultado, devido a Kučera [EPG].

Teorema III.2 – Seja um algoritmo desenvolvido para o modelo CRCW-PRAM-Forte, que utiliza tempo paralelo t e p processadores. Então este algoritmo pode ser executado numa CRCW-PRAM-Fraca em tempo t, com $O(p^2)$ processadores.

Prova – Suponha que p processadores $P_1,\ldots,P_p$ queiram, simultaneamente, escrever os valores $v_1,\ldots,v_p$ nos endereços $e_1,\ldots,e_p$, respectivamente (alguns possivelmente idênticos). Sejam $f[1\ldots p]$, $a[1\ldots p]$ e $b[1\ldots p]$ vetores auxiliares. Para cada par P_i,P_j de processadores, adicione o processador P_{ij}. Os passos abaixo se constituem num algoritmo para a simulação desejada.

Passo 1: cada processador P_i executa:

início

$a[i] := e[i]$;

$b[i] := v[i]$;

$f[i] := 0$

fim ;

Passo 2: cada processador P_{ij} executa:

se $a[i] = a[j]$ **então**

se $(b[i] > b[j])$ **ou** $(b[i] = b[j]$ **e** $i < j)$ **então** $f[j] := 1$;

Passo 3: cada processador P_i executa:

se $f[i] = 0$ **então escreva** v_i em e_i ;

Note que, após o passo 2, $f[i] = 0$ apenas para o índice i tal que $b[i] = \max_j\{b[j]\}$. ∎

O próximo teorema, cuja demonstração omitimos, é devido a Nashimi e Sahni e, independentemente, a Vishkin. Ele mostra a simulação de uma CRCW-PRAM-Forte numa EREW-PRAM. Sua demonstração pode ser vista em [EPG] e [KAR].

Teorema III.3 – Seja um algoritmo desenvolvido para o modelo CRCW-PRAM-Forte, que utiliza tempo paralelo t e p processadores. Então esse algoritmo pode ser implementado no modelo EREW-PRAM em tempo $O(t \log p)$ com p processadores. ∎

A seguir propomos uma simulação do modelo CRCW-PRAM-Fraca no modelo CREW-PRAM. Tal simulação nos será útil posteriormente, pois os algoritmos de casamento de cadeias que estudaremos foram propostos para sub-modelos de CRCW-PRAM. Enquanto que as modificações propostas por nós resultam em algoritmos que podem ser executados no modelo CREW-PRAM.

Proposição III.1 – Seja um algoritmo desenvolvido para uma máquina CRCW-PRAM-Fraca, que utiliza tempo paralelo t e p processadores. Então esse algoritmo pode ser implementado no modelo CREW-PRAM em tempo $O(t \log p)$ com p processadores.

Prova – Suponhamos que os processadores $P_1, \ldots, P_p$ queiram escrever, simultaneamente, os valores $v_1, \ldots, v_p$, respectivamente, no endereço x da memória global. Cabe à CREW-PRAM descobrir se $v_i = 0$, $\forall i$, $1 \leq i \leq p$. Caso isso aconteça, apenas um processador deve escrever o valor nulo em x. Caso contrário nada deve ser feito, pois estamos simulando o modelo CRCW-PRAM-Fraca. Em uma unidade de tempo, cada processador P_i executa a seguinte instrução , $1 \leq i \leq p$:

se $v_i = 0$ **então escreva** 0 **em** e_i **senão escreva** 1 **em** e_i ;

O vetor auxiliar $e[1 \ldots p]$ é usado da seguinte forma. Aplica-se a função *OR* sobre e, usando p processadores. O resultado é armazenado, digamos, em y. Um processador, digamos P_1, lê o valor de y. Se tal valor for nulo, então P_1 escreve 0 em x, se não for, nada a fazer. O gasto total de tempo paralelo da simulação de uma escrita concorrente é de $O(\log p)$, com p processadores (ou com $O(p/\log p)$ processadores, pelo Teorema III.1), que consiste na computação da função *OR* na CREW-PRAM. Logo, a simulação de um algoritmo de tempo paralelo t é de $O(t \log p)$. ∎

Karp e Ramachandran citam, em [KAR], o resultado apresentado por Cook, Dwork e Reischuk, onde foi provado que a computação da função *OR* numa CREW-PRAM requer tempo $\Omega(\log n)$, onde n é o tamanho da entrada, o que nos leva a crer que a simulação apresentada na Proposição III.1 seja ótima. No entanto, acreditamos que ela não o seja, pois além de não termos usado a leitura concorrente, permitida na CREW-PRAM, pode ser que haja outra simulação que não use a função *OR* e gaste menos tempo paralelo.

Apesar disso, tal simulação nos será útil, pois todos os algoritmos de casamento de cadeias que estudaremos posteriormente utilizam a computação da função *OR* quando mudam para o modelo CREW-PRAM.

Capítulo IV

Limites Inferiores

IV.1 Introdução

Muitos algoritmos para casamento de cadeias, desenvolvidos para modelos de CRCW-PRAM, têm surgido. Galil [G85] apresenta um algoritmo ótimo de tempo paralelo $O(\log m)$ para o caso do alfabeto ser de tamanho constante, onde m é o tamanho do padrão. Vishkin [V85] fornece também um algoritmo ótimo, de tempo $O(\log m)$, só que para o caso do alfabeto ser de tamanho qualquer. Breslauer e Galil [BG90] obtiveram um algoritmo ótimo de tempo paralelo $O(\log \log m)$ para um alfabeto qualquer. Recentemente Vishkin [V90] apresentou um algoritmo de tempo paralelo $O(\log^* m)$ e Galil [G92] apresentou um de tempo paralelo $O(1)$. Tais algoritmos necessitam de um pré-processamento lento do padrão , de tempo $O(\frac{\log^2 m}{\log \log m})$. Todos os algoritmos citados, além de outros, são detalhados no próximo capítulo.

Nós apresentamos neste capítulo alguns resultados com respeito a limites inferiores de tempo paralelo para o problema de casamento de cadeias. Todos os resultados foram extraídos de [BG91].

O resultado mais importante de [BG91] se constitui no limite inferior de tempo paralelo de $\Omega(\log \log m)$ para o problema de casamento de cadeias, supondo que se esteja executando m comparações em cada passo do algoritmo. Na verdade este limite inferior é também a complexidade exata para o problema, já que o algoritmo de [BG90] tem tal complexidade de tempo.

A prova de que o algoritmo de [BG90] tem essa complexidade será mostrada no capítulo V, quando detalharmos tal algoritmo.

Costuma-se classificar os algoritmos para casamento de cadeias (paralelos ou seqüenciais) de acordo com o tipo de operação envolvendo as cadeias de símbolos.

Mais precisamente, de acordo com as seguintes operações : comparação (teste de igualdade) entre dois símbolos; e operações aritméticas envolvendo inteiros resultantes de codificações feitas sobre cadeias de símbolos. Os algoritmos seqüenciais de Boyer e Moore, bem como o de KMP, por exemplo, assumem as características do primeiro tipo, enquanto que o apresentado em [MCDP] se enquadra no outro tipo. No caso paralelo, os algoritmos de [BG90] e [V85] são do primeiro tipo, enquanto que o de [G85], desenvolvido para alfabeto de tamanho fixo, é do segundo. Um algoritmo do primeiro tipo, ou seja, que depende apenas de comparações entre símbolos, é dito ser de alfabeto geral, enquanto que os demais são dito serem de alfabeto fixo. Os resultados apresentados em [BG91] são válidos para algoritmos de alfabeto geral.

Na seção IV.2 nós mostramos um limite inferior para o problema de encontrar o período de uma cadeia de símbolos. A técnica usada para isso será a mesma utilizada para mostrarmos um limite inferior para o problema de casamento de cadeias. Esse limite é mostrado na seção IV.3. Na seção IV.4 tem-se os limites inferiores para os casos onde mais de m comparações são permitidas a cada passo do algoritmo. Na última seção fazemos algumas observações importantes.

IV.2 Um Limite Inferior Para Encontrar o Período

A idéia usada para mostrar tal limite será a mesma usada para o problema geral de casamento de cadeias. Além disso, encontrar o período (definido a seguir) de um padrão consiste no pré-processamento ideal para procurá-lo num texto. Seguem algumas definições importantes.

Definição IV.1 – Sejam u e w duas cadeias de símbolos. Então u é um período de w se w é um prefixo de u^r, para algum r. Ou, equivalentemente, se w é prefixo de uw. ∎

Definição IV.2 – Uma cadeia w de símbolos é dita ser periódica se tiver um período de tamanho menor ou igual a $\frac{|w|}{2}$. ∎

Obviamente a definição IV.1 equivale a dizer que uma cadeia $S[1 \dots m]$ tem um período de tamanho k se $S[i+k] = S[i]$, para $i = 1, \dots, m-k$.

Nós mostramos uma estratégia para um adversário responder $\frac{1}{4} \log \log m$ *rounds* de m comparações , após os quais ele consegue fixar a cadeia de entrada S,

de tamanho m, de duas maneiras: numa delas S tendo um período de tamanho menor ou igual a $m/2$, portanto sendo periódica; na outra S não tendo tal período. Isto implica que qualquer algoritmo que utiliza um número menor de passos pode falhar.

A cada um dos $\frac{1}{4}\log\log m$ passos, o adversário responde a uma comparação entre $S[j]$ e $S[l]$ da seguinte forma: se $l \equiv j \ mod \ k_i$ então $S[l] = S[j]$, senão $S[l] \neq S[j]$. A cada passo i, k_i é escolhido convenientemente pelo adversário.

No início do passo i o adversário mantém um inteiro k_i, que é o tamanho de um possível período de S. O adversário responde às comparações do passo i de tal forma que algum k_{i+1} seja também o tamanho de um possível período de S. Além disso algumas posições de S são fixadas.

Seja $K_i = m^{1-4^{-(i-1)}}$. É interessante para o adversário manter as seguintes invariantes no início do passo i:

1. k_i é múltiplo de k_j, $i \geq j$;

2. $\frac{1}{2}K_i \leq k_i \leq K_i$;

3. Se $S[l]$ estava fixo então, para todo $j \equiv l \ mod \ k_i$, $S[j]$ estava fixo com o mesmo símbolo ;

4. Qualquer comparação que tenha sido respondida com diferença foi entre posições que não estavam na mesma classe módulo k_i ;

5. O número f_i de posições fixadas é tal que $f_i \leq K_i$.

Note que as invariantes 3 e 4 nos dizem que k_i é um possível período de S. Além disso, a invariante 5 nos diz que, ao final, o número de posições fixas não ultrapassa $\frac{m}{2}$.

No passo 1, com $k_1 = 1$, as invariantes obviamente valem. Nós mostraremos a seguir como o adversário deve responder às comparações do passo i e como deve escolher k_{i+1} de tal forma que as invariantes continuem valendo para o passo $i+1$.

Todos os múltiplos de k_i no intervalo $\frac{1}{2}K_{i+1} \ldots K_{i+1}$ são candidatos a k_{i+1}. Uma comparação entre $S[l]$ e $S[j]$ deve ser respondida com igualdade se $l \equiv j \ mod \ k_{i+1}$. Nós dizemos que k_{i+1} força esta comparação.

A idéia é, então , que k_{i+1} seja um candidato que force um número não muito grande de comparações , para que, ao final, poucos símbolos tenham sido fixados.

Lema IV.1 – Se $p, q \geq \sqrt{\frac{m}{k_i}}$ e são relativamente primos, então uma comparação entre $S[l]$ e $S[j]$, $l \neq j$, é forçada por no máximo um deles.

Prova – Suponha que $l \equiv j \; mod \; pk_i$ e $l \equiv j \; mod \; qk_i$, para $1 \leq l, j \leq m$. Temos então que $l \equiv j \; mod \; pqk_i$. Mas $pqk_i \geq m$ e $1 \leq l, j \leq m$. Logo $l = j$, o que nos leva a uma contradição . ∎

Definição IV.3 – Um inteiro t é dito ser primo-múltiplo de outro inteiro s se $t = p \cdot s$, onde p é primo. ∎

Lema IV.2 – O número de candidatos para k_{i+1} que são primos-múltiplos de k_i e satisfazem $\frac{1}{2}K_{i+1} \leq k_{i+1} \leq K_{i+1}$ é maior que $\frac{K_{i+1}}{4K_i \log m}$. Cada tal candidato satisfaz a condição do lema IV.1.

Prova – Estes candidatos são do tipo pk_i, com p primo. Sabemos que o número de primos entre $\frac{1}{2}x$ e x é maior que $\frac{1}{4}\frac{x}{\log x}$, para qualquer inteiro positivo x. Assim, o número de primos entre $\frac{1}{2}\frac{K_{i+1}}{k_i}$ e $\frac{K_{i+1}}{k_i}$ é, no mínimo,

$$\frac{1}{4}\frac{K_{i+1}}{k_i \log \frac{K_{i+1}}{k_i}} \geq \frac{K_{i+1}}{4K_i \log m}.$$

Por outro lado, $pk_i \geq \frac{K_{i+1}}{2}$ e

$$\frac{K_{i+1}}{2k_i} = \frac{m^{1-4^{-i}}}{2m^{1-4^{-(i-1)}}} = \frac{1}{2}m^{3 \cdot 4^{-i}} \geq \sqrt{\frac{m}{k_i}}.$$

Ou seja, cada um destes candidatos também satisfaz a condição do lema IV.1. ∎

O lema abaixo direciona a escolha de k_{i+1} por parte do adversário, de tal forma a manter a validade das invariantes para o passo $i+1$.

Lema IV.3 – Existe um candidato a k_{i+1} no intervalo $\frac{1}{2}K_{i+1} \ldots K_{i+1}$ que força no máximo $\frac{4mK_i \log m}{K_{i+1}}$ comparações .

Prova – Pelo lema IV.2, há pelo menos $\frac{K_{i+1}}{4K_i \log m}$ candidatos no intervalo

$$\frac{1}{2}K_{i+1} \ldots K_{i+1}$$

que são primos-múltiplos de k_i e satisfazem a condição do lema IV.1. Pelo lema IV.1, cada uma das m comparações é forçada por no máximo um deles. Logo, o número total de comparações forçadas por todos os candidatos é no máximo m. Daí, existe um candidato que força no máximo $\frac{4mK_i \log m}{K_{i+1}}$ comparações. ∎

Lema IV.4 – Para m suficientemente grande e $i \leq \frac{1}{4} \log\log m$,

$$1 + m^{2 \cdot 4^{-i}} 8 \log m \leq m^{3 \cdot 4^{-i}}.$$

Prova – Para m suficientemente grande,

$$\log\log(1 + 8\log m) < \frac{1}{2} \log\log m = (1 - \frac{2}{4}) \log\log m$$

$$\log(1 + 8\log m) < 4^{\frac{1}{4} \log\log m} \log m$$

$$1 + 8\log m < m^{4^{-\frac{1}{4}\log\log m}} \leq m^{4^{-i}}$$

do que segue o lema. ∎

Lema IV.5 – Suponha que as invariantes valham no início do passo i e o adversário escolha, como k_{i+1}, um candidato que força no máximo $\frac{4mK_i \log m}{K_{i+1}}$ comparações . Então o adversário pode responder às comparações do passo i de tal forma que as invariantes também valham no início do passo $i + 1$.

Prova – Depois de escolhido o tal k_{i+1} (que existe devido ao lema IV.3), para cada comparação entre $S[l]$ e $S[j]$, se $l \equiv j \ mod \ k_{i+1}$ o adversário fixa um novo símbolo para todas as posições que satisfazem esta equivalência. Comparações entre posições que já estavam fixas, o adversário responde de acordo com os próprios símbolos. Todas as comparações restantes são entre posições de classes diferentes e portanto devem ser respondidas com diferença. Obviamente, pela própria maneira de se escolher k_{i+1}, as invariantes 1 e 2 ainda valem. Mostramos agora que as outras invariantes também continuam valendo.

3. Como k_{i+1} é múltiplo de k_i, cada classe módulo k_{i+1} está na classe módulo k_i. Se uma classe módulo k_i foi fixada antes, permanece com o mesmo símbolo. Se algum símbolo foi fixado neste passo, então todas as posições pertencentes a sua classe são fixadas com o mesmo novo símbolo.

4. Classes diferentes módulo k_i permanecem diferentes módulo k_{i+1}. Então qualquer comparação que tenha sido respondida com desigualdade nos passos anteriores estão em classes diferentes módulo k_{i+1}. Todas as comparações respondidas com desigualdade neste passo são entre posições pertencentes a classes diferentes módulo k_{i+1}.

5. Vamos provar que $f_{i+1} \leq K_{i+1}$ indutivamente. São fixadas no máximo $\frac{4mK_i \log m}{K_{i+1}}$ classes módulo k_{i+1}. Há k_{i+1} tais classes e cada classe tem no

máximo $\frac{m}{k_{i+1}}$ elementos. Pelo lema 4.4, o número de elementos fixos satisfaz

$$\begin{aligned} f_{i+1} &\leq f_i + \frac{m}{k_{i+1}} \frac{4mK_i \log m}{K_{i+1}} \\ &\leq K_i[1 + (\frac{m}{K_{i+1}})^2 \; 8 \log m] \\ &\leq m^{1-4^{-(i-1)}}(1 + m^{2 \cdot 4^{-i}} \; 8 \log m) \\ &\leq m^{1-4^{-i}} = K_{i+1}. \end{aligned}$$

O que conclui o lema. ∎

Teorema IV.1 – Qualquer algoritmo paralelo, baseado em comparações , para encontrar o tamanho do período de uma cadeia $S[1 \ldots m]$ de símbolos usando m comparações em cada passo precisa de $\frac{1}{4} \log \log m$ passos.

Prova – Seja um algoritmo qualquer que encontra o período de S e seja o adversário como o descrito acima. Após $i = \frac{1}{4} \log \log m$ passos,

$$f_{i+1}, k_{i+1} \leq m^{1-4^{-\frac{1}{4}\log\log m}} \leq \frac{m}{2}.$$

O adversário tem, então , a escolha de fixar S como tendo um período de tamanho k_{i+1} (fixando cada classe módulo k_{i+1} restante com o mesmo símbolo, diferente para cada classe) ou não tendo tal período (fixando alguma classe com símbolos diferentes). Logo, qualquer algoritmo com menos de $\frac{1}{4} \log \log m$ passos pode falhar. ∎

A seguir, o teorema IV.2 estabelece um limite inferior para o problema de casamento de cadeias.

Teorema IV.2 – O limite inferior na seção anterior vale também para qualquer algoritmo paralelo de casamento de cadeias baseado em comparações.

Prova – Seja um algoritmo para casamento de cadeias. Nós apresentamos ao algoritmo um padrão $P[1 \ldots m]$ que é $S[1 \ldots m]$ e um texto $T[1 \ldots 2m-1]$ que é $S[2 \ldots 2m]$, onde $S[1 \ldots 2m]$ é uma cadeia gerada pelo adversário, da maneira como descrita acima. Após $\frac{1}{4} \log \log 2m$ passos, o adversário ainda tem a escolha de fixar S de modo a ter um período de tamanho menor ou igual a m. Neste caso teremos uma ocorrência de P em T. Ou fixar símbolos completamente diferentes às posições pertencentes a uma classe qualquer, que implica em não haver tal ocorrência. Daí, o limite inferior também vale para qualquer algoritmo de casamento de cadeias baseado em comparações , que faça m comparações a cada passo. ∎

IV.3 Permitindo Mais Comparações a cada Passo

Permitir mais comparações a cada passo significa poder aumentar o número de processadores usados pelo algoritmo.

Vamos mostrar o limite inferior para se encontrar o período de de uma cadeia $S[1\dots m]$, no caso em que $m \leq p \leq m^2$.

Teorema IV.3 – Qualquer algoritmo paralelo para encontrar o período de uma cadeia $S[1\dots m]$, baseado em comparações , usando p comparações a cada passo, $m \leq p \leq m^2$, necessita de $\Omega(\log\log_{\frac{2p}{m}} p)$ passos.

Prova – Trocamos convenientemente m por p em lugares estratégicos na prova do teorema IV.1. Em particular, fazemos $K_i = p^{1-4^{-(i-1)}}$.. Já que $k_{i+1}, f_{i+1} \leq K_{i+1}$, para concluirmos de forma análoga ao teorema 4.1, basta mostrarmos que, após $i = \log\log_{\frac{2p}{m}} p$ passos,

$$K_{i+1} = p^{1-4^{-\log\log_{\frac{2p}{m}} p}} \leq \frac{m}{2}.$$

Vamos, então , supor por absurdo que $K_{i+1} > \frac{m}{2}$. Então,

$$p^{1-4^{-\log\log_{\frac{2p}{m}} p}} > \frac{m}{2}$$

$$\frac{2p}{m} > p^{4^{-\log\log_{\frac{2p}{m}} p}}$$

$$\log\frac{2p}{m} > 4^{-\log\log_{\frac{2p}{m}} p} \log p$$

$$\frac{\log p}{\log\frac{2p}{m}} > 4^{\log\log_{\frac{2p}{m}} p}$$

$$\log_4 \left(\frac{\log p}{\log\frac{2p}{m}}\right) > \log\log_{\frac{2p}{m}} p$$

$$\frac{1}{2}\log\log_{\frac{2p}{m}} p > \log\log_{\frac{2p}{m}} p.$$

O que é uma contradição . Ou seja, $k_{i+1}, f_{i+1} \leq \frac{m}{2}$ e, portanto, $\Omega(\log\log_{\frac{2p}{m}} p)$ passos são necessários. ∎

IV.4 Algumas Considerações

Pode-se resolver o problema de casamento de cadeias em tempo constante se m^2 comparações são permitidas a cada passo, sobre uma CRCW-PRAM. O algoritmo de [BG90], de complexidade de tempo paralelo $O(\log\log m)$, pode ser executado mais lentamente se o número p de processadores usados for menor que $\frac{m}{\log\log m}$. Mais precisamente, pode ser executado em tempo paralelo $O(\frac{m}{p})$.

Em [BG91] é feita a prova de que, para $m \leq p \leq m^2$, o algoritmo de [BG90] gasta tempo paralelo $O(\log\log_{\frac{2p}{m}} p)$.

Considerando estas observações , além dos resultados apresentados nas seções anteriores, podemos derivar a complexidade paralela exata do problema de casamento de cadeias, usando p processadores, para alfabetos quaisquer. Quais sejam,

$$
\begin{array}{rll}
\Theta(\frac{m}{p}), & \text{se} & p \leq \frac{m}{\log\log m} \\
\Theta(\log\log m), & \text{se} & \frac{m}{\log\log m} \leq p \leq m \\
\Theta(\log\log_{\frac{2p}{m}} m), & \text{se} & m \leq p \leq m^2 \\
\Theta(1), & \text{se} & p \geq m^2.
\end{array}
$$

Vale a pena observarmos que todos esses limites inferiores, além de valerem apenas para o modelo CRCW-PRAM, englobam o tempo total gasto no algoritmo, incluindo portanto o tempo paralelo gasto em qualquer tipo de pré-processamento a que seja submetido o padrão .

Nota-se, portanto, que o algoritmo apresentado em [V90], assim como o algoritmo apresentado por Galil, em [G92], que usam tempo paralelo $O(\log^* m)$ e $O(1)$, respectivamente, o faz justamente porque necessitam de um pré-processamento especial, que tem complexidade de tempo paralelo $O(\frac{\log^2 m}{\log\log m})$.

Os melhores algoritmos de casamento de cadeias para o modelo CREW-PRAM desenvolvidos até hoje são aqueles feitos para o modelo CRCW-PRAM, com uma perda logarítmica na eficiência.

Capítulo V

Principais Algoritmos Paralelos

V.1 Introdução

Neste capítulo apresentaremos os algoritmos paralelos para casamento de cadeias exato considerados mais importantes.

Na seção V.2 descrevemos o algoritmo proposto por Galil [G85], que tem a restrição de funcionar apenas para o caso em que o alfabeto tem tamanho fixo. Na seção seguinte descrevemos o algoritmo de Vishkin [V85] que utiliza o então desconhecido conceito de duelo e generaliza o resultado de [G85]. Na seção V.4 mostramos o algoritmo paralelo de sufixo-prefixo [KLP] que, entre outros resultados, fornece um algoritmo ótimo para casamento de cadeias. Na seção V.5 é descrito um algoritmo paralelo devido a Breslauer e Galil [BG90], de tempo paralelo $O(\log\log m)$, que usa o fato de que, do ponto de vista de algoritmos paralelos, os duelos, inicialmente propostos em [V85], não são mais difíceis que encontrar o máximo entre n elementos. Na seção V.6 aparece o primeiro algoritmo paralelo, devido a Vishkin [V90], que precisa de um pré-processamento bastante lento do padrão. Finalmente, na seção V.7, apresentamos o trabalho de Galil [G92], onde é proposto um algoritmo paralelo de tempo constante, porém dependente, também, de um pré-processamento lento.

Os limites inferiores apresentados no capítulo anterior não valem para os algoritmos apresentados nas duas últimas seções ([V90] e [G92]), pois os limites lá provados incluem os custos com quaisquer tipos de pré-processamento do padrão.

V.2 Um Algoritmo Ótimo para Alfabeto de Tamanho Fixo

Nesta seção apresentaremos um algoritmo ótimo, devido a Zvi Galil [G85], de tempo paralelo $O(logm)$ desenvolvido para o modelo CRCW-PRAM-Fraca.

Entende-se por alfabeto de tamanho fixo aquele cujo número de símbolos é constante, independente do tamanho do texto. O algoritmo de Galil necessita dessa restrição porque precisa, num determinado ponto, compactar $\log m$ símbolos em um único número, com o custo de uma unidade de tempo.

O algoritmo é derivado de outro algoritmo, não ótimo, de tempo paralelo $t = O(\log m)$, com $p = O(n)$ processadores, desenvolvido para o caso em que n é o dobro do tamanho do padrão. Além disso faz intenso uso de propriedades concernentes à periodicidade do padrão, apesar de não fazer qualquer pré-processamento deste.

Com base nas definições IV.1 e IV.2, vamos introduzir alguns resultados referentes à periodicidade de uma cadeia de símbolos, úteis para o algoritmo. Antes, uma definição.

Definição V.1 – Sejam u e v duas cadeias tais que u é prefixo de v, sendo u periódico. Nós dizemos que a periodicidade de u continua em v se o menor período de v é o menor período de u. Caso contrário dizemos que a periodicidade termina. ∎

Lema V.1 – (*Lema da Periodicidade* (Lyndon e Schutzenberger), 1962). Se um padrão w tem períodos de tamanho P e Q, e $|x| \geq P+Q$, então w tem um período de tamanho MDC(P, Q).

Prova – Note que w tem um período de tamanho $|P-Q|$. Daí, pelo algoritmo de Euclides, é fácil ver que vale a afirmação. ∎

Lema V.2 – Se um padrão v ocorre nas posições j e $j+Q$ do texto T, $Q \leq |v|/2$, então:

(i) v é periódico, tal que Q é o tamanho de um período de v ;

(ii) v ocorre em $j+P$, onde P é o tamanho do menor período de v.

Prova –

(i) Como v ocorre em j, $u = v[1 \ldots Q]$ ocorre em j. Além disso, v ocorre em $j+Q$, logo uv ocorre em j. Ou seja, $v = T[j \ldots j+|v|-1] = uv[1 \ldots |v|-Q]$.

Portanto, segue da definição alternativa de período, que u é um período de v. Ou seja, v tem um período u tal que $|u| \leq \frac{v}{2}$.

(ii) Segue do fato V.1, já que P divide Q. ∎

Para os quatro próximos lemas, suponha que v é uma padrão periódico, tal que $v = u^k\tilde{u}$, $k > 1$, onde u é o menor período de v, $|u| = P$. Considere ainda $L = lP$, $l = \lceil\frac{|v|}{P}\rceil$.

As provas dos lemas V.3 e V.4 seguem diretamente de uma simples contagem de períodos.

Lema V.3 – Se v ocorre nas posições j e $j + qP$ do texto, $q \leq k$, então $u^{k+q}\tilde{u}$ ocorre em j. ∎

Lema V.4 – v ocorre nas posições j, $j + P$, e $j + L$ do texto se, e somente se, $u^{k+l}\tilde{u}$ ocorre em j. ∎

Lema V.5 – Se v ocorre nas posições j e $j + \Delta$, $\Delta \leq |v| - P$, então Δ é um múltiplo de P.

Prova – Suponha que Δ não seja um múltiplo de P. Então $\Delta = qP+r$, $0 < r < P$ e $q < k$. Seja $w = u^{(k-q)}\tilde{u}$. w é um sufixo de de v, logo w ocorre em $j + qP$. w também é um prefixo de v, daí ocorre em $j + \Delta = j + qP + r$. Pelo lema V.2, w tem um período de tamanho r e, pelo fato V.1, w tem um período de tamanho $\text{MDC}(P, r) < P$. O que contradiz a hipótese de P ser o tamanho do menor período de v. ∎

Definição V.2 – Uma ocorrência de um padrão em um texto na posição j é dita ser importante se v não ocorre em $j + P$. ∎

Lema V.6 – Se há duas ocorrências importantes de v em r e s, $r > s$, então $r - s > |v| - P$.

Prova – Suponha que $r - s \leq |v| - P$. Pelo fato V.5, $r - s = qP$. Pelo fato V.4, $u^{(k+q)}\tilde{u}$ ocorre em s e portanto v ocorre em $s + P$. O que contradiz a hipótese de haver uma ocorrência importante na posição s. ∎

O algoritmo, como dissemos anteriormente, é derivado de um outro algoritmo, este não ótimo, que funciona para o caso em que o tamanho do texto é o dobro do tamanho do padrão. A este chamaremos de algoritmo básico.

A entrada para o algoritmo básico é uma cadeia $z = x\$y$, de tamanho $3m+1$, onde $|x| = m$ e $|y| = 2m$, sendo x o padrão e y o texto. Além disso, ambas x e y são cadeias cujos símbolos pertencem a um alfabeto Σ, tal que $|\Sigma|$ é fixo, ou seja, independe de m. Mais ainda, $\$ \notin \Sigma$.

A saída é um vetor booleano $MATCH$, de tamanho $3m+1$, de tal forma que, ao final da execução do algoritmo, $MATCH[j] = 1$ se, e somente se, uma ocorrência de x começa em $z[j]$.

A idéia consiste em considerar prefixos cada vez maiores de x e, a cada passo, encontrar todas as ocorrências desse prefixo em z. Em y por razões óbvias e em x para determinar suas periodicidades.

O algoritmo tem $\lceil \log m \rceil$ passos. Após o passo i, $MATCH[j] = 1$ se, e somente se, o prefixo $x^{(i)}$ de x , de tamanho 2^i, ocorre em j.

Seja $x^{(i+1)} = x^{(i)}v^{(i)}$. O passo $i+1$ consiste em verificar se cada ocorrência de $x^{(i)}$ em j é seguida de uma ocorrência de $v^{(i)}$. Se a resposta a este teste for negativa, então o algoritmo faz $MATCH[j]$ receber o valor nulo.

O algoritmo pode se encontrar em um dos dois estados seguintes: estado periódico e estado aperiódico. Dependendo do estado em que o algoritmo se encontra, executa um conjunto diferente de instruções .

No passo $i+1$, se $x^{(i)}$ é periódico, então o algoritmo entra no estado periódico. Caso contrário, entra no estado aperiódico.

Este controle é feito da seguinte maneira. No passo $i+1$, $MATCH$ é dividido em blocos de tamanho 2^{i-1}. Se $MATCH[j] = 0$, para $2 \leq j \leq 2^{i-1}$ (obviamente $MATCH[1] = 1$ sempre), então pelo lema V.2 $x^{(i)}$ não é periódico e, portanto, o algoritmo entra no estado aperiódico.

Durante a execução do algoritmo, os processadores precisam obviamente se comunicar. Para uma comunicação global são usadas duas variáveis: G_1, que contém o tamanho do período, caso o algoritmo esteja no estado periódico e 0 caso contrário; e G_2, que contém o valor de $L = lP$, onde $l = \lceil \frac{2^i}{P} \rceil$, para cada passo i. Além disso, para uma comunicação local entre os processadores responsáveis por um bloco, cada um (bloco) terá uma variável, chamada H, que *apontará* para o único 1 neste bloco. Os H's serão usados somente nos passos em que o algoritmo estiver no estado aperiódico.

Vamos agora supor que o algoritmo básico esteja no passo $i+1$ e no estado aperiódico.

Neste caso o primeiro bloco de $MATCH$ tem apenas um único 1, obviamente na posição 1 de z. A certificação deste fato pode ser feita pelo processador responsável pelo segundo H do estágio i (o primeiro do estágio $i+1$). Ele olha para o seu H. Se está vazio, então é porque o algoritmo se encontra no estado aperiódico.

No passo $i+1$, dizemos que a *propriedade i vale* se cada bloco tem no máximo um 1.

No estado aperiódico, além do primeiro bloco conter apenas um 1, os demais blocos contém no máximo dois 1's. Neste caso, o 1 mais a esquerda de cada bloco que contém dois 1's deve ser eliminado. Em outras palavras, para cada tal índice j, $MATCH[j]$ recebe o valor nulo. Esta ocorrência de $x^{(i+1)}$ não pode acontecer na posição j, pois está acontecendo na posição $j + \Delta$, para algum Δ, $0 < \Delta \leq 2^{i-1}$, e o primeiro bloco não tem um 1 na posição $\Delta + 1$. Logo, $x^{(i+1)}$ não pode ocorrer em j.

Note que, a cada passo, entre dois blocos vizinhos no passo anterior, digamos $2k - 1$ e $2k$, apenas um sobrevive. Então a operação de eliminação do primeiro 1 num bloco com dois 1's pode ser feita pelo processador responsável pelo bloco sobrevivente.

Como resultado, a propriedade i vale. Então cada grupo de 2^{i-1} processadores responsável por um bloco que contém um 1 executa o que chamaremos de *operação regular.*

A operação regular, que consiste em testar se uma ocorrência de $x^{(i)}$ em j é seguida de uma ocorrência de $v^{(i)}$, pode ser feita da seguinte forma. O r-ésimo processador do bloco que contém um 1 executa as comparações

$$x[2^i + r + \Delta] =^? z[j + 2^i + r - 1 + \Delta], \quad \Delta \in \{0, 2^{i-1}\}.$$

Se um dos 2^{i-1} processadores do bloco obtém uma resposta negativa, então faz $MATCH[j] := 0$. Este é o único ponto do algoritmo básico em que, de fato, ocorre escrita concorrente. Note que o tipo de escrita concorrente caracteriza o fato de se estar usando o modelo PRAM-CRCW-Fraca.

Suponha agora que o algoritmo básico se encontra no passo $i + 1$, no estado periódico.

No caso particular em que o algoritmo estava no caso aperiódico no passo anterior, como $MATCH[1] = 1$, o processador responsável pelo segundo H do passo i (o primeiro do passo $i + 1$) olha para o seu H. Se este tem valor não nulo, então contém $P + 1$, ou seja, $x^{(i)}$ é periódico, com menor período de tamanho P. Então este processador executa $G_1 := P$.

Durante os passos nos quais o algoritmo se encontra no estado periódico, nenhum H é usado. Por outro lado, G_1 conterá P e G_2 conterá $L = lP$, onde $l = \lceil \frac{2^i}{P} \rceil$.

Seja $\hat{x}^{(i+1)}$ o prefixo de x de tamanho $2^i + L$. Temos então que

$$|x^{(i+1)}| = 2^{i+1} \leq |\hat{x}^{(i+1)}| < 2^{i+1} + P.$$

Usa-se esse prefixo de x, ao invés de $x^{(i+1)}$, por conveniência, devido ao lema V.4.

Para testar se a periodicidade de $x^{(i)}$ continua em $\hat{x}^{(i+1)}$ usa-se o lema V.4 da seguinte forma. Faz-se $v = x^{(i)}$, $j = 1$, $\hat{x}^{(i+1)} = u^{k+l}\tilde{u}$. O primeiro processador testa se $MATCH[P + 1] = 1$ e $MATCH[L + 1] = 1$ (note que o primeiro teste é redundante no caso em que o algoritmo vem do estado aperiódico). Caso ambos

sejam iguais a 1, significa que a periodicidade de $x^{(i)}$ continua em $\hat{x}^{(i+1)}$. Resta agora encontrar todas as ocorrências de $\hat{x}^{(i+1)}$ em z. Isto é feito como segue. Pelo lema V.4 e fazendo $v = x^{(i)}$, $u^{k+l}\tilde{u} = \hat{x}^{(i+1)}$, o processador p_j, tal que $MATCH[j] = 1$, testa se $MATCH[P + j] = 1$ e $MATCH[L + j] = 1$. Se um deles é igual a zero, então p_j faz $MATCH[j]$ valer zero.

No teste da continuidade da periodicidade de $x^{(i)}$ em $\hat{x}^{(i+1)}$, o processador p_1 olha para $MATCH[P + 1]$. Se é igual a zero (significa que foi alterado no passo anterior), então a ocorrência de $x^{(i)}$ em 1 é importante e o processador p_1 faz G_1 valer zero. Caso contrário, olha para $MATCH[L + 1]$. Se é igual a 1, a periodicidade continua. Caso contrário cada processador p_j, $1 \leq j \leq L + 1 - P$ testa se existe uma ocorrência importante em j. O único processador que consegue escrever faz G_1 valer $j - 1$. Isto acontece porque, no caso da periodicidade não continuar e, como $MATCH[1] = 1$, um entre $MATCH[P + 1]$ e $MATCH[L + 1]$ é zero. Logo, no mínimo uma das ocorrências de $x^{(i)}$ nas posições $j \leq L + 1 - P$ é importante. Pelos lemas V.5 e V.6, ou a ocorrência em 1 é importante ou existe exatamente uma ocorrência importante em algum j, $1 \leq j \leq L + 1 - P$.

Dado o valor de $G_1 = j - 1$, cada processador p_r tal que $MATCH[r] = 1$ checa se existe uma ocorrência importante de $x^{(i)}$ em $r + j - 1$. Isto pode ser feito usando-se o vetor $MATCH$. Como existe uma ocorrência importante de $x^{(i)}$ em $x^{(i+1)}$, na posição j, caso exista tal ocorrência importante em $r + j - 1$, deve ser eliminada. Como resultado, propriedade i vale e, portanto, executa-se a operação regular da mesma forma como definida anteriormente, com a diferença de que, neste caso, deve-se antes alterar os valores antigos dos H's dos blocos que contém um 1. Cada processador p_r com $MATCH[r] = 1$ escreve r em seu H.

No caso em que a periodicidade continua, é preciso ainda alterar o valor de L em G_2 para o próximo passo. Isto é facilmente feito como segue. Se $2\,L - P > 2^{(i+1)}$, então $L := 2L - P$, senão $L := 2L$.

Antes de discutirmos a complexidade do algoritmo básico, precisamos descrever seu passo inicial, assim como o final, em virtude destes serem diferentes dos demais, dados os detalhes que os envolvem.

No passo inicial ($i = 1$), o processador p_j testa se $z[j]z[j + 1] = x[1]x[2]$. Se o teste for positivo, então p_j faz $MATCH[j]$ valer um e faz o j-ésimo H apontar, no segundo passo, para a posição j. O segundo passo começa com os blocos tendo tamanho 1 normalmente, como descrito acima.

Quando o algoritmo entra no último passo no estado aperiódico, ou no estado periódico sendo que a periodicidade termina, basta executar a chamada operação regular. Mesmo no caso em que $L + 2^i > m$, ou seja, mesmo quando $|\hat{x}^{(i+1)}| > |x|$. A operação regular é feita até o m-ésimo símbolo apenas.

O que pode acontencer é que, ao entrar no último passo, $MATCH[j] = 1$ para algum j tal que $j + 2^{i+1} > m + 1$, então $x^{(i+1)}$ não pode ocorrer em j, pois é muito longo e $\$ \notin \Sigma$. A única mudança ocorre quando se entra no último passo no estado periódico e, além disso, a periodicidade de $x^{(i)}$ continua em $x^{(i+1)}$.

Seja uma *ocorrência especial* de $x^{(i)}$ em j uma ocorrência de $x^{(i)}$ em j tal que $j+P+2^i > m+1$, ou seja, a próxima ocorrência de $x^{(i)}$ em $j+P$ é tal que seu símbolo mais a direita está numa posição j', tal que $j' > m$ (isto pode acontecer, já que os testes de ocorrência são feitos com base no vetor $MATCH$ e nos lemas anteriores).

Obviamente uma ocorrência especial de $x^{(i)}$ em j, além de ser única, é tal que $j = \alpha P + 1$, para algum α. Ou seja, $x = u^\alpha u^{\alpha'} \tilde{u} u'$, para algum α', onde $\tilde{u}u'$ é um prefixo de u^2. Então cada processador p_r, tal que $MATCH\ [\ r\ ]\ =\ 1$, testa se $MATCH[r+j-1] = 1$. No caso negativo é porque x não ocorre em r. O valor de $MATCH[r]$ deve ser alterado. No caso positivo, p_r testa se $MATCH[r+j-1+P] = 1$. Se isto acontecer é porque x ocorre em r. Caso contrário nada podemos afirmar. Por outro lado, neste caso, a ocorrência de $x^{(i)}$ em $r+j-1$ é importante. Então, pelo lema V.6, se restringirmos a atenção às posições r, tais que a ocorrência de $x^{(i)}$ em $r+j-1$ é importante, a propriedade i vale. Aplica-se então, com a devida reativação dos H's, a operação regular para testar se tais ocorrências de $x^{(i)}$ são seguidas de x', onde $x^{(i)}x' = x$.

Como dissemos anteriormente, o algoritmo básico tem $\lceil \log m \rceil$ passos. Cada passo no estado periódico é constituído das seguintes operações, cada uma delas de tempo paralelo constante: o teste da periodicidade de $x^{(i)}$; o teste da continuação de tal periodicidade; algumas mudanças particulares em $MATCH$; e a operação regular. Com exceção da primeira e da última, que são feitas em duas etapas,de tempo constante, as demais operações são feitas diretamente sobre o vetor $MATCH$.

Já um passo no estado aperiódico tem um custo ainda menor. O teste da periodicidade é o mesmo para o outro estado. O mesmo acontece para a operação regular. O teste da validade da propriedade i é feito com base nos H's do passo anterior, já que lá, com certeza, o algoritmo se encontrará no estado aperiódico.

Todas operações em cada passo são de tempo paralelo constante. Portanto, o algoritmo básico pode ser executado em tempo paralelo $O(\log m)$, com m processadores.

Do algoritmo básico, descrito acima, deriva-se um algoritmo ótimo para casamento de cadeias. Na realidade, trata-se da execução do algoritmo básico com apenas $O(\frac{m}{\log m})$ processadores, usando uma espécie de truque, utilizado para multiplicar matrizes booleanas no algoritmo denominado *Four Russians Algorithm* [AHU].

Tal truque consiste em compactar $\log m$ símbolos em um número. Divide-se z e $MATCH$ em blocos de s símbolos consecutivos, onde $s = c \log m$ e c depende do tamanho do alfabeto Σ. Cada processador p_r se responsabiliza por $z[j]$ e $MATCH[j]$, para $j \in A_r = (r-1)s+1, \cdots, rs$.

Primeiro, cada processador p_r compacta cada cadeia de z começando em $z[j]$, $j \in A_r$, de tamanho s. Esta operação de compactação de símbolos leva tempo paralelo $O(1)$. Então p_r compara cada novo símbolo $\hat{z}[j]$, $j \in A_r$, com $\hat{z}[1]$ e, se eles são iguais, faz $MATCH[j] = 1$. estas operações têm o mesmo efeito dos primeiros

$\lfloor \log s \rfloor$ passos do algoritmo básico e tomam tempo paralelo $O(s) = O(\log m)$.

Assume-se que no $(t+1)$-ésimo passo o algoritmo se encontra no estado aperiódico. Os próximos passos seguem normalmente, como no algoritmo básico. Nos passos de estado aperiódico os símbolos $\hat{z}[j]$ são usados. Omitimos, propositalmente, alguns detalhes de tal implementação.

Com a adaptação proposta acima para o algoritmo básico, tem-se uma família de algoritmos com custo $O(m)$, para $p \leq \frac{m}{\log m}$ processadores.

Além disso, deve-se considerar o caso em que $|x|$ e $|y|$ não estão relacionados entre si.

Pode-se resolver esta questão da seguinte forma. seja $n = |x| + |y|$. Se $p \leq \frac{2n}{m}$, então divide-se y em $p/4$ partes iguais de tamanho $\frac{4y}{p}$. Cada processador ficará responsável pela cadeia resultante da concatenação da i-ésima com a $(i+1)$-ésima partes, onde procurará por todas as ocorrências de x, em tempo $O(n/p)$, pois $m \leq 2 \cdot \frac{4|y|}{p} = O(\frac{n}{p})$.

Se $p > \frac{2n}{m}$, quebra-se y em partes sobrepostas de tamanho $2m$, de tal forma que o início de uma parte distancie m do início da parte seguinte. O número h de tais partes é $O(\frac{|y|}{m})$. Distribui-se, então, h processadores por parte. Então todas as ocorrências de x numa parte podem ser encontradas num tempo paralelo t, tal que $t \cdot (p/s) = O(m)$, ou seja, $t \cdot p = O(m \cdot s) = O(|y|) = O(n)$.

Ao concluirmos esta seção, convém lembrarmos que o algoritmo proposto em [G85], pode ser executado em tempo paralelo $O(\log^2 m)$, com $O(\frac{m}{\log^2 m})$ processadores numa CREW-PRAM, já que o único ponto onde de fato existe escrita concorrente é na operação regular, onde 2^{i-1} processadores responsáveis por um bloco computam o resultado da aplicação de uma função AND, e sabemos que tal operação pode ser feita em tempo paralelo $O(\log d)$, onde d é o tamanho da entrada para a função.

V.3 Uma Nova Técnica – Os Duelos

Nesta seção descrevemos o trabalho apresentado por Vishkin [V85], onde é proposto um algoritmo paralelo ótimo, de tempo paralelo $O(\log m)$, para o modelo CRCW-PRAM.

O algoritmo proposto por Vishkin generaliza o resultado de Galil (apresentado na seção anterior), no sentido de que usa algumas idéias lá preconizadas. Por outro lado, propõe uma técnica nova e interessante de eliminação de posições do texto onde pretensamente o padrão ocorreria. Essa técnica, chamada de *duelo*, funciona com base em informações referentes à distância entre duas possíveis

ocorrências. Essas duas posições se duelam de tal forma que no máximo uma sobrevive. Tal técnica será formalmente descrita oportunamente.

Duas etapas principais formam o algoritmo de Vishkin. A primeira faz uma análise do padrão. Especificamente, descobre se o padrão é ou não periódico. Caso seja, determina o tamanho P de seu menor período. A outra etapa encontra todas as ocorrências do padrão (já pré-analisado) no texto.

Na subseção V.3.1 apresentamos uma caracterização para a periodicidade do padrão. Na subseção seguinte mostramos como é feita a análise do padrão. Na subseção V.3.3 apresentamos o algoritmo que busca no texto o padrão pré-processado. Na subseção V.3.4 falamos sobre a complexidade de todo o algoritmo.

V.3.1 Uma caracterização para a periodicidade

Vamos agora mostrar uma caracterização para a periodicidade de um padrão, com base nas definições IV.1 e IV.2.

Seja o padrão $PAT[1 \cdots m]$. Suponha que $PAT[j \cdots m]$ não seja uma prefixo de PAT, para algum j, $2 \leq j \leq m$. Então certamente existe um inteiro w, $1 \leq w \leq m - j + 1$, tal que $PAT[w] \neq PAT[j + w - 1]$. Veja a figura V.1. Nós dizemos que w é uma *testemunha* para um não-emparelhamento. Note que w é uma testemunha da não existência de um período de tamanho $P = j - 1$.

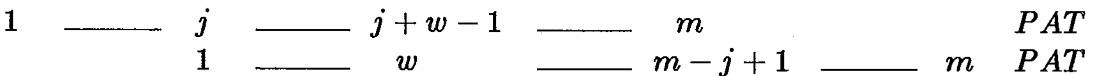


Figura V.1

A determinação de tais testemunhas será dada pelo vetor WIT, de tamanho m, de tal forma que

$$WIT[j] \quad = \quad \begin{cases} w & \text{se } \exists w,\ 1 \leq w \leq m - j + 1, \\ & \text{tal que } PAT[w] \neq PAT[j + w - 1] \\ 0 & \text{caso contrário} \end{cases}$$

Podemos então dizer que o padrão PAT é aperiódico se, e somente se, $WIT[j] \neq 0$, para cada j, $2 \leq j \leq \frac{m}{2} + 1$. Note que $WIT[1] = 0$ sempre.

Logo, a descoberta da periodicidade de PAT, ou seja, a análise do padrão, resume-se em calcular os valores de $WIT[j]$, $2 \leq j \leq \frac{m}{2} + 1$. Tal análise é feita na próxima subseção.

V.3.2 A análise do padrão

Como foi visto na subseção anterior, o pré-processamento do padrão consiste em calcular $WIT[j]$, $2 \leq j \leq \frac{m}{2} + 1$ (lembre-se que $WIT[1] = 0$ e $WIT[j]$, com $j > \frac{m}{2} + 1$, não nos interessa).

A idéia da análise do padrão feita por Vishkin consiste em descobrir as periodicidades de cada prefixo de PAT, fazendo isso para prefixos cada vez maiores (de maneira parecida àquela feita no algoritmo de Galil, visto na seção V.2).

No início, $WIT[j] = 0$, para todo j, $1 \leq j \leq m$. Isto é, $PAT[j \cdots m]$ é suspeito ser um prefixo de PAT. A análise tem $O(\lfloor \log m \rfloor)$ passos. Após o passo k, as três seguintes propriedades devem estar satisfeitas:

(i) **k-certeza:** Para todo j, $1 \leq j \leq 2^k$, tem-se que $WIT[j] = 0$ se, e somente se, $PAT[1 \cdots 2^{k+1} - j + 1] = PAT[j \cdots 2^{k+1}]$. Ou seja, $WIT[j]$ está calculado corretamente "até um certo ponto". Em outras palavras, se $2^{k+1} \geq m$, então $WIT[1 \cdots 2^k]$ tem os valores finais desejados para o vetor WIT.

(ii) **k-dispersão:** Seja um k-bloco uma parte de WIT de tamanho 2^k, de tal forma que existam $\lceil \frac{m}{2^k} \rceil$ k-blocos disjuntos em WIT. Ou seja, os k-blocos são $WIT[1 \cdots 2^k], WIT[2^k + 1 \cdots 2 \cdot 2^k], \ldots WIT[t \cdot 2^k + 1 \cdots (t+1) \cdot 2^k]$. Se o primeiro k-bloco de WIT tem apenas um zero (obviamente na posição 1), então cada k-bloco de WIT tem no máximo um zero. Obviamente, justifica-se o nome da propriedade, o fato de se ter zeros dispersos em WIT.

(iii) **k-limitação:** $WIT[j] \leq 2^{k+1}, 1 \leq j \leq m$.

O algoritmo de análise do padrão, assim como o algoritmo de Galil apresentado na seção anterior, pode estar em um dos dois estados: estado aperiódico e estado periódico.

Seja $ZERO(k, a)$ a posição mais a esquerda no a-ésimo k-bloco de WIT tal que $WIT[ZERO(k, a)] = 0$, $1 \leq a \leq \lceil \frac{m}{2^k} \rceil$, ou uma indicação (do tipo "vazio") de que não existe tal zero. Estas variáveis serão usadas nos passos onde o algoritmo se encontra no estado aperiódico. Já nos passos k onde o algoritmo estiver no estado periódico, será usada a variável global $PERIODO(k)$, que será igual ao tamanho do menor período de $PAT[1 \cdots 2^{k+1}]$.

A análise do padrão PAT começa, como dissemos anteriormente, fazendo-se $WIT[j]$ valer 0, $\forall j$, $1 \leq j \leq m$. Testa-se, então, se $PAT[1] \neq PAT[2]$. Caso isso aconteça, entra-se no passo 1 no estado aperiódico (sabe-se aqui que PAT não tem período de tamanho 1). Caso contrário, entra-se no passo 1 no estado periódico, pois, como $PAT[1] = PAT[2]$, suspeita-se que PAT tenha um período de tamanho 1.

Inicialmente faz-se $ZERO(0,a) := a$, $\forall a$, $1 \leq a \leq m$, já que $WIT[j] = 0$, $\forall j$, $1 \leq j \leq m$.

Suponha agora que o algoritmo de análise do padrão entre no passo $k+1$ no estado aperiódico. Neste instante, k-certeza e k-dispersão são satisfeitas. Além disso, $WIT[2 \cdots 2^k]$ não tem zeros. A seguinte seqüência de comandos é executada:

se $ZERO(k,2) \neq$ *vazio*
{ *o segundo k-bloco de WIT tem um zero?* }
  então
    para todo j, $1 \leq j \leq 2^{k+2} - ZERO(k,2) + 1$
      faça em paralelo *(k+1)-certeza*
        se $PAT[j] \neq PAT[ZERO(k,2) + j - 1]$
          então $WIT[ZERO(k,2)] := j$;
se $WIT[ZERO(k,2)] = 0$
  então
    início
      $PERIODO(k+1) := ZERO(k,2) - 1$;
      "vá para o passo $k+2$ no estado periódico"
    fim
  senão
    início
      "satisfaça $(k+1)$-dispersão";
      "vá para o passo $k+2$ no estado aperiódico";
    fim

Na realidade, o que se pretende fazer com os comandos acima é descobrir se existe suspeita de periodicidade do prefixo de PAT de tamanho 2^{k+1}.

Vamos agora descrever como pode ser satisfeita a propriedade $(k+1)$-dispersão. Neste instante, $(k+1)$-certeza está satisfeita. Mais ainda, $WIT(j) \neq 0$, $2 \leq j \leq 2^{k+1}$.

Suponha que dois k-blocos vizinhos $2a-1$ e $2a$ tenham, cada um, um zero (todos os k-blocos tem no máximo um zero, já que k-dispersão vale). Sejam j_1 e j_2 posições tais que $WIT[j_1] = WIT[j_2] = 0$, onde j_1 e j_2 estão, respectivamente, nos k-blocos $2a-1$ e $2a$. Isto significa que PAT ocorre em $PAT[j_1]$ e $PAT[j_2]$. Veja a figura V.2. Mas isto não pode acontecer, pois $2 \leq j_2 - j_1 + 1 \leq 2^{k+1}$ e, pelo que foi dito no parágrafo anterior, $WIT(j_2 - j_1 + 1) = w \neq 0$. Ou seja, $PAT[j_2 - j_1 + w] = PAT[j_2 + w - 1] = PAT[w]$. Mas como $WIT[j_2 - j_1 + 1] = w \neq 0$ temos que $PAT[w] \neq PAT[j_2 - j_1 + 1]$.

Aqui se aplica o conceito de duelo. Duela-se as duas posições j_1 e j_2 de tal forma que no máximo uma delas sobrevive.

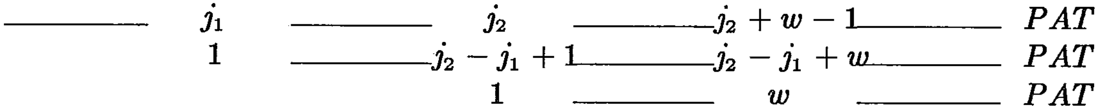


Figura V.2

As instruções abaixo, que servem para satisfazer a propriedade $(k+1)$-dispersão, são executadas, em paralelo, para cada $(k+1)$-bloco a, $2 \leq a \leq \lceil \frac{m}{2^{k+1}} \rceil$. Note que o primeiro $(k+1)$-bloco não necessita deste processamento, pois sabemos que ele tem apenas um único zero (na posição 1).

```
se ZERO(k, 2a) = vazio
  então ZERO(k + 1, a) := ZERO(k, 2a − 1)
  senão
    se ZERO(k, 2a − 1) = vazio
      então ZERO(k + 1, a) := ZERO(k, 2a)
      senão
        início
          o novo (k+1)-bloco tem 2 zeros, em j1 e j2
          j1 := ZERO(k, 2a − 1) ;
          j2 := ZERO(k, 2a) ;
          w := WIT[j2 − j1 + 1] ;
          se PAT[j2 − j1 + w] ≠ PAT[j2 + w − 1]
            então WIT[j1] := j2 − j1 + w ;
          se PAT[w] ≠ PAT[j2 + w − 1]
            então WIT[j2] := w ;
          se WIT[j1] = 0
            então ZERO(k + 1, a) := j1
            senão
              se WIT[j2] = 0
                então ZERO(k + 1, a) := j2
                senão ZERO(k + 1, a) := vazio
        fim
```

A figura V.2 ilustra perfeitamente o objetivo das instruções acima, após as quais, o algoritmo de análise do padrão inicia o passo $k+2$ no estado aperiódico.

Suponha agora que o mesmo algoritmo inicia o passo $k+1$ no estado periódico. Suponha ainda que o último passo no qual o algoritmo se encontrava no estado aperiódico foi o passo $k'+1$. Então, neste ponto, k-certeza e k'-dispersão valem. Além disso, suspeita-se que $PAT[1\cdots 2^{k+1}]$ tenha um período de tamanho P.

Existe a possibilidade de haver posições j no primeiro $(k+1)$-bloco tais que $WIT[j]=0$ e $j-1$ não é divisível por P, o que contradiz o fato de $PAT[1\cdots 2^{k+1}]$ ter um período de tamanho P. Deve-se eliminar tais posições. Como $(k+1)$-certeza vale, significa que as referidas posições estão no segundo k-bloco. As instruções abaixo eliminam este tipo de problema.

para todo j, $2^k \le j \le 2^{k+1}$ **faça em paralelo**
  se $WIT[j]=0$ **e** $(j \bmod P) \ne 1$
    então
      início
        $w := WIT[j \bmod P]$;
        se $PAT[j-1+w] \ne PAT[w]$ **então** $WIT[j] := w$;
      fim ;

É preciso agora verificar se a periodicidade de $PAT[1\cdots 2^{k+1}]$ continua no prefixo $PAT[1\cdots 2^{k+2}]$. Caso continue, o algoritmo inicia o passo $k+2$ no estado periódico. Caso contrário, é porque algum símbolo numa posição j, $2^{k+1} < j \le 2^{k+2}$, causou a mudança em $WIT[P+1]$. Tal símbolo é um contra-exemplo também para o caso em que algum múltiplo de P, menor que 2^{k+1}, seja um período de $PAT[1\cdots 2^{k+2}]$. Ou seja, os valores de WIT nas posições do tipo $jP+1$, $2 \le j \le (2^{k+1}-1)/P$, devem ser confirmados. Resta apenas a confirmação do valor de WIT para as posições i, $2^k < i \le 2^{k+1}$ que, após todos estes testes, ainda tem $WIT[i]=0$. É possível mostrar que existem no máximo quatro tais posições (essa prova pode ser vista em [V85]). Se alguma dessas posições restantes ainda ficar com seu WIT igual a zero, o algoritmo altera o valor de $PERIODO(k+1)$. Neste ponto $(k+1)$-certeza está satisfeita. Deve-se agora satisfazer k-dispersão. Se $WIT[2^k+1\cdots 2^{k+1}]$ tiver um zero, então o algoritmo passa para o passo $k+2$ no estado periódico. Caso contrário, satisfaz $(k+1)$-dispersão e vai para o passo $k+2$ no estado aperiódico.

As instruções a seguir implementam os objetivos expostos no parágrafo anterior.

para todo j, $2^{k+1} < j \leq 2^{k+2}$ **faça em paralelo**
  se $PAT[j] \neq PAT[j \bmod P]$ **então** $WIT[P+1] := j - P$;
se $WIT[P+1] = 0$ *P pode ser período de* $PAT[1 \cdots 2^{k+2}]$
  então "comece o passo $k+2$ no estado periódico" ;
  senão início
    para todo j, $2 \leq j \leq (2^{k+1} - 1)/P$
      faça em paralelo $WIT[jP+1] := WIT[P+1] - (j-1)P$;
    para todo i, $2^k \leq i \leq 2^{k+1}$ **tal que** $WIT[i] = 0$ **faça**
      início
        para todo j, $1 \leq j \leq 2^{k+2} - i + 1$
          faça em paralelo
            se $PAT[j] \neq PAT[i-1+j]$ **entao** $WIT[i] := j$;
        se $WIT[i] = 0$ **então** $PERIODO(k+1) := i - 1$;
      fim
    "satisfaça k-dispersão" ;
    se $PERIODO(k+1) \neq$ *vazio*
      então "comece o passo $k+2$ no estado periódico"
      senão
        início
          "satisfaça k-dispersão" ;
          "comece o passo $k+2$ no estado aperiódico" ;
        fim
    fim

A instrução "satisfaça k-dispersão", a partir da validade de k'-dispersão pode ser feita em $k - k'$ iterações. Na iteração t, $1 \leq t \leq k - k'$, $(k' + t)$-dispersão é satisfeita. Cada iteração é idêntica ao modo de como é satisfeita normalmente $(k+1)$-dispersão, vista anteriormente.

Em nenhum momento preocupou-se em satisfazer a propriedade k-limitação. Na verdade, ela sempre vale, ao final do passo k (veja a prova em [V85]). A preocupação maior em validá-la está no fato de que esta propriedade não deixa a corretude do algoritmo de análise do padrão ser afetada, quando se tem referências a algum índice j de PAT tal que $j > m$. O algoritmo, portanto, pode proceder como se $PAT[j]$ fosse igual a qualquer símbolo do alfabeto.

O último passo do algoritmo de análise do padrão também está relacionado com a propriedade k-limitação. Ela permite que o algoritmo, com apenas um teste,

saiba exatamente quando parar e como executar o último passo. Os detalhes da implementação do último passos podem ser vistos em [V85].

A complexidade da análise do padrão é vista da seguinte forma. No início do passo k, no estado aperiódico, necessita-se de $O(2^k)$ operações e tempo $O(1)$. Além disso, para a satisfação da propriedade k-dispersão, são necessárias $O(1)$ operações e tempo $O(1)$ para cada um dos $O(\lceil \frac{m}{2^k} \rceil)$ k-blocos. Ou seja, $O(\frac{m}{2^k})$ operações e tempo $O(1)$. No passo k, no estado periódico, precisa-se de $O(2^k)$ operações e tempo $O(1)$. Como o algoritmo de análise do padrão tem um total de $O(\log m)$ passos, tem-se, pelo teorema de Brent, a existência de um algoritmo de tempo $O(\log m)$, com $m/\log m$ processadores de uma CRCW-PRAM.

V.3.3 A procura do padrão pré-analisado

Dado que o vetor $WIT[1 \cdots m]$, ou pelo menos $WIT[1 \cdots \frac{m}{2} + 1]$, está completo, podemos efetivamente procurar o padrão $PAT[1 \cdots m]$ no texto $T[1 \cdots n]$. Tal algoritmo depende diretamente do fato de PAT ser ou não periódico e tem, como saída, o vetor $MATCH[1 \cdots n]$.

Para se descobrir quais as posições i do texto T nas quais PAT ocorre, no caso de PAT ser aperiódico, usa-se exatamente a idéia dos duelos entre posições de T.

O lema V.7, a seguir, é indispensável para o uso do duelo na eliminação de algumas posições do texto, com base no fato de que, no caso aperiódico, $WIT[j] \neq 0, 2 \leq j \leq \frac{m}{2} + 1$.

Lema V.7 – Sejam i e j duas posições distintas do texto T tais que $j - i \leq m/2$. Se PAT é aperiódico, então PAT ocorre no máximo em uma dessas duas posições.

Prova – Suponha, por absurdo, que PAT ocorra em i e j e seja $w = WIT[j-i+1]$ (veja a figura V.3). Como $2 \leq j - i + 1 \leq \frac{m}{2} + 1$, tem-se que $w \neq 0$, já que PAT é aperiódico. Logo, $PAT[j - i + w] = T[j + w - 1] = PAT[w]$. Mas $PAT[w] \neq PAT[j - i + w]$, pela própria definição de WIT. Ou seja, PAT pode ocorrer em i ou em j, mas não em ambas. ∎

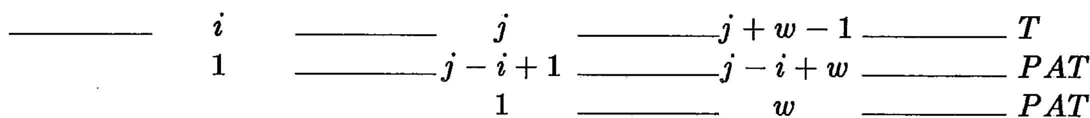


Figura V.3

O lema acima noz diz que duas ocorrências de um padrão aperiódico de tamanho m devem estar distantes de pelo menos $\frac{m}{2}+1$. Assim, em um passo, pode-se eliminar pelo menos uma das duas posições.

Assuma, por simplicidade, que ambos m e $n' = n-m+1$ sejam potências de 2. Divida sucessivamente o conjunto $\{1, 2, \cdots, n'\}$ de índices em blocos disjuntos de tamanho 2^k, como na subseção anterior, para $k = 0, \cdots, \log m - 1$. Ou seja, $T[1 \cdots n']$ tem os seguintes $\lceil \frac{n'}{2^k} \rceil$ k-blocos: $[1 \cdots 2^k]$, $[2^k + 1 \cdots 2 \cdot 2^k]$, ...

Como $k \leq \log m - 1$, temos que cada k-bloco tem no máximo $2^{\log m - 1} = m/2$ índices. Logo, dadas duas posições i e j num mesmo k-bloco, com certeza $j-i+1 \leq m/2$ e, portanto, $j - i \leq m/2$. Ou seja, i e j se enquadram no enunciado do lema V.7, para $k \leq \log m - 1$. Em cada k-bloco haverá no máximo uma posição onde PAT ocorre. Tal posição será chamada de posição candidata. Ao executarmos o comando $k := k+1$ teremos em cada k-bloco duas posições candidatas. Duela-se então as duas posições de cada k-bloco em paralelo. Assim, cada k-bloco terá novamente apenas uma posição candidata. Este passo é executado para $k = 1, \ldots, \log m - 1$. O duelo entre duas posições i e j será determinado pela função $DUELO(i, j, VETOR)$. Tal função tem a responsabilidade de, para cada k-bloco, eliminar uma das duas posições candidatas, fazendo com que à posição candidata r, morta no duelo, seja atribuída a instrução $VETOR[r] := 0$. No início, $MATCH[j] = 1$, $\forall j$, $1 \leq j \leq n'$. Depois, as chamadas da função $DUELO$ são feitas para o vetor $MATCH$.

O duelo entre duas posições candidatas segue exatamente a idéia contida na demonstração do lema V.7. Sejam i e j duas posições candidadtas em um k-bloco. Seja ainda $t = j-i+1$ e $w = WIT[t]$ (lembre-se que $w \neq 0$, pois $t = j-i+1 \leq \frac{m}{2}+1$ e PAT é aperiódico). No máximo uma das duas afirmações abaixo é verdadeira:

(i) $T[j+w-1] = PAT[j-i+w]$

(ii) $T[j+w-1] = PAT[w]$

já que $PAT[j-i+w] \neq PAT[w]$. A função $DUELO(i, j, VETOR)$ testa se $PAT[j+w-1] \neq PAT[w]$. Se isto acontecer, é porque PAT não ocorre em j. Logo a posição i ganha o duelo e j é eliminada. Repare, neste caso, que não sabemos se PAT ocorre em i. Só temos, neste instante, a certeza de que PAT não ocorre em j. Não há problema quanto a isso, pois o fato da posição i ter ganho o duelo, significa apenas que i é uma posição candidata para a próxima iteração e será eliminada mais cedo ou mais tarde, caso PAT não ocorra em i. Agora, se (i) vale, então j vence o duelo.

Ao final, quando $k = \log m - 1$, temos $\frac{n'}{2^{\log m - 1}} = O(n/m)$ posições candidatas. Neste ponto checamos, para cada uma das $O(n/m)$ tais posições r, se PAT ocorre em r.

A seguir, a descrição da função $DUELO(i,j,VETOR)$. A variável $VETOR$ é usada porque nós utilizaremos esta mesma função posteriormente. Aqui ela será chamada tendo como parâmetro o vetor $MATCH$.

```
FUNÇÃO DUELO(i,j,VETOR) ;
início
  w := WIT[j - i + 1] ;
  se T[j + w - 1] ≠ PAT[w]
    então
      início
        DUELO := i ;
        VETOR[j] := 0
      fim
    senão
      início
        DUELO := j ;
        VETOR[i] := 0
      fim
fim ;
```

Toda essa computação tem a estrutura de uma árvore. Quando $k = 1$, duas posições são candidatas, para cada 1-bloco. Como a função $DUELO$ elimina uma das duas posições candidatas em cada k-bloco, podemos garantir que, ao final, teremos exatamente $2n'/m$ posições candidatas.

Perceba ainda que, como $j \leq n' = n - m + 1$ e $w = WIT[j - i + 1] \leq m - 1$, temos que $j + w - 1 < n$. Ou seja, o índice $j + w - 1$, que é usado na função $DUELO$, está dentro dos limites dos índices de T.

Para armazenarmos os resultados dos duelos nos diversos passos da computação, usaremos uma função f auxiliar, tal que: para cada k, $1 \leq k \leq \log m - 1$, tenhamos:

$$f(k,j) := DUELO(f(k-1, 2j-1), f(k-1, 2j)), \quad 1 \leq j \leq n/2^k.$$

Os valores de f são calculados gradativamente, com o aumento de k. No início, $f(0,j) := j$, $1 \leq j \leq n$.

Descrevemos, a seguir, o passo que finaliza o algoritmo para o caso aperiódico. Este passo possui três etapas. A primeira inicializa o vetor $MATCH$, a segunda elimina, através da função duelo, algumas posições t fazendo $MATCH[t] := 0$. A terceira testa, símbolo por símbolo, se de fato PAT ocorre em t.

para todo j, $1 \leq j \leq n$ **faça em paralelo**
  se $j \leq n'$ **então** $MATCH[j] := 1$ **senão** $MATCH[j] := 0$;
para todo j, $1 \leq j \leq n$ **faça em paralelo** $f(0,j) := j$;
para $k := 1$ **até** $\log m - 1$ **faça**
  para todo j, $1 \leq j \leq n'/2^k$, **faça em paralelo**
   início
    $p := f(k-1, 2j-1)$;
    $q := f(k-1, 2j)$;
    p e q são duas candidatas no j-ésimo bloco
    $f(k,j) := DUELO(p,q)$
   fim
para todo j, $1 \leq j \leq n'$, **tal que** $MATCH[j] = 1$
  faça em paralelo
   "Teste, símbolo por símbolo, se PAT ocorre em j";
   Se não ocorre faça $MATCH[j] := 0$

A primeira etapa pode ser executada em tempo $O(\log m)$ com $O(n'/\log m)$ processadores. A complexidade da segunda etapa pode ser medida pelo número de execuções da função $DUELO$, já que a mesma tem custo constante. O número total de duelos é igual a

$$\left(\frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{\log m - 1}}\right) = O(n).$$

Logo, como esta etapa tem profundidade $O(\log m)$, podemos executá-la neste tempo paralelo com $O(n/\log m)$ processadores. A última etapa tem um total de $(\frac{2n'}{m}) \cdot m = O(n)$ operações de teste. Pode, então, ser executada em tempo $O(\log m)$, com $O(n/\log m)$ processadores. Nenhuma das etapas necessita de escrita concorrente.

Conclusão: Este passo, que finaliza o algoritmo para o caso aperiódico, pode ser executado em tempo $O(\log m)$ com $O(n/\log m)$ processadores de uma CREW-PRAM.

O caso periódico, ou seja, o caso em que PAT é periódico, é resolvido como descrito a seguir, a partir do valor de P, determinado na análise do padrão.

Digamos que $PAT = u^s v$, onde u é o menor período de PAT e $|u| = P$. Daí, $s = \lfloor \frac{m}{P} \rfloor$ e $|v| = m \bmod P$ $(|v| < |u|)$.

Como v é prefixo de u, a ocorrência de $PAT' = uv$ numa posição i qualquer é perfeitamente compatível com a ocorrência de PAT' em $i + P$.

Seja i uma posição de T onde PAT' ocorre. Vamos definir $LARGEST[i]$ como sendo o número de vezes que PAT' ocorre, a partir de i, de P em P posições. Ou seja, se PAT' ocorre em i,$i + P$,...,$i + kP$ e não ocorre em $i + (k + 1)P$, então $LARGEST[i] = k + 1$. Fica claro, a partir dessa idéia, que PAT ocorre em i se, e somente se, $LARGEST[i] \geq s$. Vamos, então, não calcular exatamente $LARGEST[i]$, para cada posição i onde ocorre PAT', mas apenas determinar quais as posições i do texto são tais que $LARGEST[i] \geq s$. As *principais* ocorrências de PAT' em T serão dadas pelo vetor $LISTA[1 \cdots n]$, tal que $LISTA[i] = 1$ se PAT' ocorre em i e zero caso contrário. O termo *ocorrências principais* usado aqui será explicado logo a seguir. Antes, três resultados importantes.

Lema V.8 – Seja PAT um padrão periódico, com menor período de tamanho P, tal que PAT ocorre em duas posições distintas i e j do texto T. Então $|i - j| \geq P$.

Prova – Suponhamos, por contradição, que $|i - j| < P$. Digamos que $i < j$. Então $j - i + 1 \leq P$. Como P é o tamanho do menor período de PAT, $P + 1$ é tal que $WIT[P + 1] = 0$ e nenhum outro índice r, $2 \leq r < P + 1$, de WIT, é tal que $WIT[r] = 0$. Logo, como $j - i + 1 \leq P$, temos que $WIT[j - i + 1] = w \neq 0$. Ou seja, $PAT[w] \neq PAT[j - i + 1 + w - 1] = PAT[j - i + w]$. Mas, por outro lado, como PAT ocorre em i, temos que $PAT[j - i + w] = T[j + w - 1]$ e, como PAT ocorre em j, temos que $PAT[w] = T[j + w - 1]$. tal contradição prova o lema. ▮

Note que o lema acima é o correspondente ao lema V.7 para o caso periódico.

Lema V.9 – Seja PAT uma padrão periódico e u, onde $|u| \leq |PAT|/2$, um período de PAT. Suponha que u também seja periódico, tal que $u = r^k$, $k > 1$. Então r é também um período de PAT.

Prova – PAT é um prefixo de u^s, $s > 1$. Então PAT é prefixo de r^{ks}. ▮

Lema V.10 – Seja $PAT = u^s v$, $s > 1$, um padrão periódico, tendo u como menor período, de tamanho P. Então u é o menor período de $w = u^2$.

Prova– Suponhamos por contradição que w tenha um período $\bar{u}$ tal que $|\bar{u}| = \bar{P} < P$. Então, pelo lema da periodicidade (lema V.1), w tem um período de tamanho $P_0 = \text{mdc}(P, \bar{P})$, já que $P + \bar{P} < 2P = |w|$. Então, como P é múltiplo de P_0, temos que u deve ser do tipo $(u_1 u_2 \cdots u_{P_0})^{P/P_0}$. Logo, pelo lema V.9, $u_1 \cdots u_{P_0}$ é período de PAT. Absurdo, pois $P_0 < P$ e P é o tamanho do menor período de PAT. ∎

Com o objetivo de construir o vetor $LISTA$, vamos inicializá-lo com as ocorrências de u^2 em T.

Pelos lemas V.8 e V.10, dadas duas posições i e j em T, $i < j$ e $j - i < P$, u^2 pode ocorrer em i ou em j, mas nunca em ambas. Isto nos leva a usar novamente a função $DUELO$, vista anteriormente. Desta vez fazemos $\lceil \log P \rceil$ *rounds* de duelos, garantindo, ao final, que cada $\lceil \log P \rceil$-bloco tenha no máximo um índice onde u^2 pode ocorrer. Daí testamos, símbolo por símbolo, para cada posição r suspeita de ocorrência de u^2 se $u^2 v$ ocorre em r. Como ao final temos $O(\frac{n}{P})$ posições suspeitas e cada verificação de ocorrência de $u^2 v$ custa $|u^2 v| < 3P = O(P)$ testes, o custo total é de $O(n)$ testes, símbolo a símbolo. Isto pode ser feito em tempo $O(\log m)$ com $O(n/\log m)$ processadores.

Até aqui $LISTA[i] = 1$ se, e somente se, $u^2 v$ ocorre em i. Na verdade, fazemos os duelos apenas entre as posições $1,\ldots,n'$ de T. Fazemos isto por dois motivos, quais sejam, o fato de que certamente $u^2 v$ não ocorre em k, com $k > n'$, e também o fato de que um duelo envolve posições i e j tais que $j \leq n'$. Consequentemente, $j + w - 1 \leq n' + m - 2 < n$, já que $w = WIT[j] \leq m - 1$.

Basta agora, para cada i, $1 \leq i \leq n$, tal que $LISTA[i] = 1$, fazermos $LISTA[i + P] := 1$, pois, se $u^2 v$ ocorre em i, então uv ocorre em $i + P$. É muito importante observarmos que pode haver casos em que uv ocorre em r mas $LISTA[r] = 0$. isto pode acontecer porque estamos inicialmente a procura de $u^2 v$ e não de uv. por outro lado isso não é um problema, pois uma ocorrência isolada de uv em r não nos interessa, visto que $PAT = u^s v$ e $s > 1$. Ou seja, uma ocorrência de PAT acontece somente onde $u^2 v$ ocorre. Em outras palavras, ao final teremos $LISTA[i] = 1$ se, e somente se, PAT' ocorre não isoladamente em i. Essas são as ditas ocorrências importantes mencionadas anteriormente.

Como dissemos, não precisamos calcular o valor exato de $LARGEST[i]$. Basta sabermos se, a partir de i, o número de ocorrências de uv, de P em P posições, é maior ou igual a s.

Calcularemos progressivamente, por *recursive doubling* ([GRY88]) o possível valor de $LARGEST[i]$, em $\log m$ passos. Ao final, como $s \leq m$ e a profundidade da computação é de $\log m$, se $LARGEST[i] < m$, então teremos o valor exato de $LARGEST[i]$. Caso contrário, $LARGEST[i] \geq m$ e então saberemos que PAT ocorre em i.

Para conseguirmos a complexidade desejada no cálculo de $LARGEST[i]$, vamos dividir o vetor $LISTA$ em P sub-listas disjuntas. Cada sub-lista l com as

posições $l + kP$ $(k \geq 0)$ de $LISTA$, onde $1 \leq l \leq P$. Cada sub-lista terá tamanho $O(n/P)$. Basta contarmos, com *recursive doubling*, a partir de cada ocorrência de um 1, quantos 1's consecutivos há em cada sub-lista, até no máximo m, ($\log m$ passos). Isto pode ser feito em tempo $O(\log m)$, com $O(\frac{n/P}{\log m})$ processadores para cada sub-lista. Ou seja, para o total de P sub-listas, o tempo gasto será de $O(\log m)$, com o uso de $O(\frac{n/P}{\log m} \cdot P) = O(n/\log m)$ processadores. As inicializações podem ser feitas com a mesma complexidade.

Para facilitar a compreensão, vamos dividir o algoritmo de busca de PAT em T, com PAT periódico em três etapas como segue. Na etapa 1 o padrão u^2 é analisado e u^2v é encontrado em T. na etapa 2, a partir do vetor $LISTA$ construído na etapa 1, $LARGEST[i]$ é estimado e, finalmente, na etapa 3 o vetor $MATCH$ é construído de forma direta.

De novo, usamos a função auxiliar f para armazenarmos os resultados dos duelos, como fizemos no caso aperiódico.

A seguir, as instruções necessárias para a implementação das idéias acima. Tais instruções encerram o algoritmo no caso periódico.

início (ETAPA 1)

"Fazer a análise do padrão $u^2 = PAT[1 \cdots 2P]$" ;

para todo j, $1 \leq j \leq n$, **faça em paralelo**
se $j \leq n'$
então $LISTA[j] := 1$
senão $LISTA[j] := 0$;

"Faça $\lceil \log P \rceil$ rounds de duelos, como
na segunda etapa do algoritmo do caso aperiódico"

para todo j, $1 \leq j \leq n'$, **tal que** $LISTA[j] = 1$
faça em paralelo
início
"Teste a ocorrência de u^2v em j" ;
Se u^2v ocorre em j então $LISTA[j + P] := 1$
fim

fim (ETAPA 1)

início (ETAPA 2)

```
para todo k, 1 ≤ k ≤ ⌈n/ log m⌉
  faça em paralelo
    para j := (k − 1) log m + 1 até k log m faça
      LARGEST[j] := LISTA[j] ;
"Decomponha LARGEST como descrito acima, em P sub-listas" ;
"Seja LL[1 ··· l] uma sub-lista, onde l = ⌊n/P⌋" ;
"Adicione a ela o elemento LL[l + 1] = 0, por conveniência" ;

para todo k, 1 ≤ k ≤ ⌈l/ log m⌉
  faça em paralelo
    para j := k log m − 1 até (k − 1) log m + 1 faça
      se LL[j] ≠ 0 então LL[j] := LL[j + 1] + 1 ;

para todo k, 1 ≤ k ≤ ⌈l/ log m⌉
  faça em paralelo
  para j := (k − 1) log m + 1 até k log m faça
    se LL[j] ≠ 0
      então NEXT[j] := LL[j] + j
      senão NEXT[j] := j ;

execute log m vezes
para todo k, 1 ≤ k ≤ ⌈l/ log m⌉ faça em paralelo
  início
    j := (k − 1) log m + 1 ;
    se NEXT[j] ≠ NEXT[NEXT[j]] então
      início
        LL[j] := LL[j] + LL[NEXT[j]] ;
        NEXT[j] := NEXT[NEXT[j]]
      fim
  fim
```

```
para todo k, 1 ≤ k ≤ ⌈l/ log m⌉ faça em paralelo
  início
    j := k log m ;
    se (LL[j + 1] ≠ 0) e (LL[j] ≠ 0) então
      início
        LL[j] := LL[j + 1] + 1 ;
        j := j − 1 ;
        enquanto LL[j] ≠ 0) e (j > (k − 1) log m + 1) faça
          início
            LL[j] := LL[j] + 1 ;
            j := j − 1 ;
          fim
      fim
  fim ;
"Recomponha LARGEST diretamente a partir das sub-listas LL" ;


fim (ETAPA 2)

início (ETAPA 3)
  para todo k, 1 ≤ k ≤ ⌈n/ log m⌉
    faça em paralelo
    para j := (k − 1) log m + 1 até k log m faça
      se LARGEST[j] ≥ s
        então MATCH[j] := 1
        senão MATCH[j] := 0
fim (ETAPA 3)
```

A análise do padrão u^2 leva tempo paralelo $O(\log m)$ com $O(n/\log m)$ processadores. Os $\lceil \log P \rceil$ rounds de duelos somam, como no caso aperiódico, um total de $O(n)$ chamadas da função $DUELO$, com profundidade $\lceil \log P \rceil = O(\log m)$. Podem, então, ser executados em tempo $O(\log m)$ com $O(n/\log m)$ processadores. Restam, então, $O(n/P)$ posições candidatas. Cada verificação de ocorrência de $u^2 v$ em uma posição suspeita pode ser feita em tempo $O(\log P)$, com $O(P/\log P)$ processadores. Como temos $O(n/P)$ tais posições, gastaremos um tempo total de $O(\log P)$ com $O(\frac{n}{P} \cdot \frac{P}{\log P}) = O(n/\log P) = O(n/\log m)$ processadores, já que $P = O(m)$.

Como explicado anteriormente, a etapa 2, que consiste no cálculo estimado de $LARGEST$, pode ser executada em tempo paralelo $O(\log m)$ com $O(n/\log m)$ processadores. A etapa 3 obviamente tem a mesma complexidade.

V.3.4 Complexidade

A parte da busca de PAT em T de Vishkin foi modificada nesta apresentação, de tal forma que escritas concorrentes não fossem necessárias. Em outras palavras, esta parte do algoritmo pode ser executada em tempo paralelo $O(\log m)$, com $\frac{m}{\log m}$ processadores de uma CREW-PRAM.

Por outro lado, a análise do padrão obtém uma perda natural de tempo paralelo na passagem para o modelo CREW-PRAM. Por esse motivo, talvez, o autor não tenha se preocupado em fazer tais mudanças nesta parte do algoritmo.

No capítulo VI deste trabalho, propomos um algoritmo para a análise do padrão para o modelo CREW-PRAM, que funciona para o caso em que o alfabeto é de tamanho fixo e $m = O(\log^2 n)$. Para esses casos, nosso algoritmo tem a mesma complexidade que o de Vishkin, porém, sem escrita concorrente.

V.4 Uma Aplicação de um Algoritmo de Sufixo-prefixo

Nesta seção apresentaremos um algoritmo ótimo para casamento de cadeias, resultante de uma aplicação direta de um algoritmo também ótimo para o problema de sufixo-prefixo, onde, dadas duas cadeias A e B, de mesmo tamanho, deve-se encontrar quais os sufixos de A são iguais aos prefixos de B. Este resultado é devido a Kedem, Palem e Landau, proposto em [KLP].

Além do algoritmo ótimo para casamento de cadeias, deriva-se do algoritmo para sufixo-prefixo algoritmos para vários outros problemas, variantes do problema original de casamento de padrões.

Na subseção V.4.1 introduzimos o problema de sufixo-prefixo. Na subseção seguinte apresentamos um algoritmo não ótimo para o mesmo problema. Na subseção V.4.3 descrevemos o algoritmo ótimo e, finalmente, na subseção V.4.4,

apresentamos o algoritmo ótimo para casamento de cadeias derivado do algoritmo ótimo para sufixo-prefixo.

V.4.1 O problema de sufixo-prefixo

O problema de sufixo-prefixo consiste em determinar, dadas $A = a_0 a_1 \cdots a_{m-1}$ e $B = b_0 b_1 \cdots b_{m-1}$, duas cadeias de símbolos pertencentes a um alfabeto $\Sigma \subseteq \{0, \cdots, m-1\}$, se o sufixo de A, $a_{m-i-1} a_{m-i} \cdots a_{m-1}$ é igual ao prefixo $b_0 b_1 \cdots b_i$ de B. A saída é um vetor de bits $\delta[0 \cdots m-1]$, onde

$$\delta[i] = 1 \quad \text{se, e somente se,} \quad a_{m-i-1} a_{m-i} \cdots a_{m-1} = b_0 b_1 \cdots b_i.$$

O algoritmo mais trivial para resolver este problema é o que testa a igualdade para cada sufixo de A e cada prefixo de B de mesmo tamanho. Obviamente esta técnica tem um custo muito alto: $O(m^2)$.

Se imaginarmos que o conjunto de sufixos e prefixos (m sufixos de A e m prefixos de B) podem ser distribuídos entre, no máximo, $2m$ classes de equivalência (sobre a relação de igualdade), então podemos reduzir tal ineficiência. Precisamos, portanto, de uma função que mapeie este conjunto em $\{0, \cdots, 2m-1\}$, com a propriedade de que duas cadeias de mesmo tamanho sejam iguais se, e somente se, pertencerem à mesma classe de equivalência.

Seja S o conjunto formado pelos sufixos de A e prefixos de B. A função

$$\chi : S \rightarrow \{0, 1, \cdots, 2m-1\}$$

será chamada de *característica* e fornecerá diretamente os valores de δ. Além disso, assumiremos que $\log m$ divide m.

V.4.2 Um algoritmo não ótimo para a função característica

Apresentaremos agora um algoritmo que, apesar de não ser ótimo para a computação da função característica, apresentada na subseção anterior, mostra a idéia básica para o algoritmo ótimo para sufixo-prefixo.

O algoritmo tem duas etapas. A primeira etapa determina, num total de $O(m \log m)$ operações, as características de algumas sub-cadeias de A e B. Na segunda etapa, determina em tempo $O(\log m)$ as características dos prefixos de B e sufixos de A.

A primeira etapa é executada em $\log m + 1$ iterações. Para cada iteração $i = 0, 1, \cdots \log m$, computa-se a característica de todas as sub-cadeias de A e B de tamanho 2^i, usando informações da iteração $i - 1$. É necessário que todas as cadeias de mesmo tamanho sejam processadas concorrentemente.

Seja $a_j a_{j+1} \cdots a_{j+2^i-1}$ uma sub-cadeia de A (o mesmo vale para as sub-cadeias de B). Assuma que a característica $\chi(a_j a_{j+1} \cdots a_{j+2^{i-1}-1})$ e também a característica $\chi(a_{j+2^{i-1}} a_{j+2^{i-1}+1} \cdots a_{j+2^i-1})$ já tenham sido calculadas na iteração $i-1$. Além disso, estejam no conjunto $\{0, 1, \cdots, 2m-1\}$. A determinação de $\chi(a_j a_{j+1} \cdots a_{j+2^i-1})$ é simples. O processador P_k escreve k na posição

$$\chi(a_j \cdots a_{j+2^{i-1}-1}) + 2m\chi(a_{j+2^{i-1}} \cdots a_{j+2^i-1})$$

de um vetor T de $(2m)^2$ posições. Então P_k executa a instrução

$$\chi(a_j \cdots a_{j+2^i-1}) := T(\chi(a_j \cdots a_{j+2^{i-1}-1}) + 2m\chi(a_{j+2^{i-1}} \cdots a_{j+2^i-1})).$$

Já que a posição descrita acima reflete um número escrito na base $2m$, podemos garantir que dois processadores escreverão no mesmo local se, e somente se, estiverem processaando sub-cadeias iguais.

Note que o modelo de computação paralela usado deve ser do tipo CRCW-PRAM-forte ou CRCW-PRAM-prioridade (neste caso de escrita concorrente são equivalentes). Além disso, temos que $\chi(a_j \cdots a_{j+2^i-1}) \in \{0, 1, \cdots, 2m-1\}$, já que $k \in \{0, 1, \cdots, 2m-1\}$.

O custo da primeira etapa é $O(m \log m)$, dada por

$$\sum_{i=0}^{\log m - 1} m - 2^i + 1 = (m+1)\log m - m + 2,$$

onde $m - 2^i + 1$ é o número de sub-cadeias de tamanho 2^i numa cadeia de tamanho m.

Na segunda etapa, as características calculadas acima são combinadas para se determinar as características dos sufixos de A e dos prefixos de B. Em $\log m$ passos podemos calcular $\chi(\bar{a})$ e $\chi(\bar{b})$, onde $\bar{a}$ e $\bar{b}$ são, respectivamente, sufixo de A e prefixo de B. Suponha que $|\bar{a}| = \sum_{j=0}^{\log m} c_j 2^j$, $c_j \in \{0, 1\}$ (o mesmo vale para $\bar{b}$). $\bar{a}$ pode ser escrito na forma $\bar{a}_0 \bar{a}_1 \cdots \bar{a}_{\log m}$, onde $|\bar{a}_j| = c_j 2^j$. No passo i, $i = 1, \ldots, \log m$, $\chi(\bar{a}_0 \cdots \bar{a}_i)$ pode ser obtido combinando $\chi(\bar{a}_0 \cdots \bar{a}_{i-1})$ e $\chi(\bar{a}_i)$. Todos os $\bar{a}$ e $\bar{b}$ de mesmo tamanho devem ser processados simultaneamente.

Apesar da segunda fase poder ser executada em tempo paralelo $O(\log m)$, seu custo também é $O(m \log m)$.

Uma alternativa seria diminuir o custo para determinar a característica dos prefixos de B, da seguinte forma. Na primeira etapa, de passos $i = 0, 1, \ldots, \log m$, computa-se, para cada passo i, a característica de todas as sub-cadeias do conjunto $A_1 = \{a_j \cdots a_{j+2^i-1} | 0 \leq j \leq m - 2^i\}$ de A e de todas as sub-cadeias $B_1 = \{b_j \cdots b_{j+2^i-1} | 0 \leq j \leq m - 2^i$ e j divisível por $2^i\}$. Note que $|A_1| = m - 2^i + 1$ e $|B_1| = \frac{m}{2^i}$. Ao final, portanto, o número total de operações para B diminuiu para $O(m)$, já o número total de operações para A continua $O(m \log m)$.

Na segunda etapa, esta tendo $\log m$ passos, são computadas, para cada passo i, $i = \log m - 1, \ldots, 0$, as características das sub-cadeias de A pertencentes ao conjunto $\{a_j \cdots a_{k+j} | k+j \leq m-1$, $j+1$ divisível por 2^i e não divisível por $2^{i+1}\}$. São computadas ainda as características das subcadeias de B pertencentes ao conjunto $\{b_0 \cdots b_j | j > 2^i, j+1$ divisível por 2^i e não divisível por $2^{i+1}\}$. Neste ponto, tanto as características das sub-cadeias de A quanto as de B são calculadas com base nas características determinadas nos passos anteriores, ou até em algum passo da primeira etapa, da mesma forma como é feito na segunda fase do algoritmo anterior, em tempo $O(1)$.

O escalonamento proposto para o cálculo das características dos sufixos de B é interessante, mas infelizmente não se aplica à cadeia A, já que seus sufixos maiores não compartilham os sufixos menores na simulação.

O custo deste algoritmo alternativo, portanto, também é $O(m \log m)$.

V.4.3 Um algoritmo ótimo para sufixo-prefixo

Com base na dificuldade de fazer com que A se enquadrasse de forma eficiente na simulação descrita na seção anterior, os autores de [KLP] propuseram um algoritmo ótimo para o problema de sufixo-prefixo.

Tal algoritmo baseia-se na idéia de dividir ambas as cadeias A e B em sub-cadeias sucessivas de tamanho $\log m$. Suponha que já estejam computadas as características das sub-cadeias $b_i b_{i+1} \cdots b_{i+\log m-1}$, $i = 0, 1, \ldots, m - \log m$ e $a_i a_{i+1} \cdots a_{i+\log m-1}$, i divisível por $\log m$ (no final desta subseção explicaremos como é feita tal computação eficientemente).

Sejam $\bar{b}_i = \chi(b_i b_{i+1} \cdots b_{i+\log m-1})$ e $\bar{a}_i = \chi(a_i a_{i+1} \cdots a_{i+\log m-1})$. Sejam

ainda

$$\begin{aligned}
\bar{A} &= \bar{a}_0 \bar{a}_{\log m} \bar{a}_{2\log m} \cdots \bar{a}_{m-\log m} \quad , \\
\bar{B}_1 &= \bar{b}_1 \bar{b}_{\log m+1} \bar{b}_{2\log m+1} \cdots \bar{b}_{m-2\log m+1} \quad , \\
\bar{B}_2 &= \bar{b}_2 \bar{b}_{\log m+2} \bar{b}_{2\log m+2} \cdots \bar{b}_{m-2\log m+2} \quad , \\
&\vdots \\
\bar{B}_{\log m} &= \bar{b}_{\log m} \bar{b}_{2\log m} \bar{b}_{3\log m} \cdots \bar{b}_{m-\log m} .
\end{aligned}$$

Seja $j \in \{1,2,\ldots,m\}$, $j = c_1 \log m + c_2$, onde $c_1 \geq 0$ e $1 \leq c_1 \leq \log m$. Além disso, $i = j - 1$. Então não é difícil verificarmos que $\delta[i] = 1$ se, e somente se,

$$\begin{aligned}
&(a) \quad a_{m-c_1\log m} a_{m-c_1\log m+1} \cdots a_{m-1} = b_{c_2} b_{c_2+1} \cdots b_i \quad , \text{ e} \\
&(b) \quad a_{m-i-1} a_{m-i} \cdots a_{m-c_1+\log m-1} = b_0 b_1 \cdots b_{c_2-1} .
\end{aligned}$$

Mais ainda, não é difícil verificar que (a) é equivalente a:

$$\chi(\bar{a}_{m-c_1\log m} \bar{a}_{m-(c_1-1)\log m} \cdots \bar{a}_{m-\log m}) = \chi(\bar{b}_{c_2} \bar{b}_{c_2+\log m} \cdots \bar{b}_{c_2+(c_1-1)\log m}).$$

A condição (a) é, na verdade, um teste de igualdade entre o sufixo de $\bar{A}$, de tamanho c_1 (de tamanho $c_1 \log m$ em A) é igual ao prefixo de tamanho c_1 em $\bar{B}_{c_2}$, que é a sub-cadeia de tamanho $c_1 \log m$ de B que começa em b_{c_2}. A condição (b) complementa o teste. A idéia principal está no fato de que restam apenas $\log m$ sub-cadeias distintas para se calcular as características. Já em A restam $\frac{m}{\log m}$ conjuntos de subcadeias, cada conjunto contendo sufixos de sub-cadeias de tamanho $\log m$.

Antes de vermos a descrição completa, bem como a complexidade do algoritmo, vamos ver como se calcula a característica das sub-cadeias de tamanho $\log m$ de A e B.

Lembre-se que precisamos calcular as $\frac{m}{\log m}$ características

$$\bar{a}_0, \bar{a}_{\log m}, \ldots, \bar{a}_{m-\log m}$$

de A e as $m - \log m$ características $\bar{b}_0, \bar{b}_1, \ldots, \bar{b}_{m-\log m}$.

Consideremos primeiramente as $\frac{m}{\log m}$ sub-cadeias de A. Como são de tamanho pequeno ($\log m$), é interessante e eficiente se construir uma máquina de reconhecimento para elas. Isto pode ser feito usando o algoritmo de construção de um autômato de reconhecimento proposto em [ACO] (descrito brevemente no capítulo II). O novo símbolo resultante de cada sub-cadeia seria seu estado de aceitação. As funções g, de transição, e f, de falha, funcionariam de forma equivalente.

O número total de operações, necessário para construção, em paralelo, do autômato é de $O(\frac{m}{\log m} \cdot \log m) = O(m)$ (igual a soma dos tamanhos dos padrões a serem reconhecidos).

Agora consideremos as $m - \log m$ sub-cadeias de B, que obviamente não podem ser processadas da mesma forma com que foram processadas as $\frac{m}{\log m}$ sub-cadeias de A.

O processador P_k $(0 \leq k \leq \frac{m}{\log m})$ computa as características

$$\bar{b}_{k \log m}, \bar{b}_{k \log m + 1}, \ldots, \bar{b}_{(k+1) \log m - 1}$$

das $\log m$ sub-cadeias

$$\begin{array}{l} b_{k \log m} \cdots b_{(k+1) \log m - 1} \quad , \\ b_{k \log m + 1} \cdots b_{(k+1) \log m} \quad , \\ \vdots \\ b_{(k+1) \log m - 1} \cdots b_{(k+2) \log m - 2} . \end{array}$$

Executa-se o autômato para as sub-cadeias de A seqüencialmente sobre o "texto" $b_{k \log m} \cdots b_{(k+2) \log m - 2}$ seqüencialmente. Como a execução seqüencial do autômato gasta tempo $O(n)$, onde n é o tamanho do texto, e cada sub-cadeia $b_{k \log m} \cdots b_{(k+2) \log m - 2}$ tem tamanho $O(\log m)$, temos que o tempo gasto por cada processador é $O(\log m)$.

Vamos agora descrever, passo a passo, o algoritmo ótimo para sufixo-prefixo.

1. Construa, em paralelo, o autômato M (de [ACO]) que aceite as seguintes $\frac{m}{\log m}$ cadeias:

$$\begin{array}{l} a_0 a_1 \cdots a_{\log m - 1} \quad , \\ a_{\log m} a_{\log m + 1} \cdots a_{2 \log m - 1} \quad , \\ \vdots \\ a_{m - \log m} a_{m - \log m + 1} \cdots a_{m - 1} \quad . \end{array}$$

   Seja $\bar{a}_i$, i múltiplo de $\log m$, o nome (ou número) do estado de aceitação de $a_i a_{i+1} \cdots a_{i + \log m - 1}$.

   Crie a cadeia $\bar{A} = \bar{a}_0 \bar{a}_{\log m} \cdots \bar{a}_{m - \log m}$.

   OBS.: O passo 1 toma tempo paralelo $O(\log m)$ e um total de $O(m)$ operações.

2. Execute, em paralelo, M tendo B como texto. Como resultado, crie a cadeia $\bar{B} = \bar{b}_0\bar{b}_1 \cdots \bar{b}_{m - \log m}$, onde $\bar{b}_i$ é o estado de aceitação de $b_i b_{i+1} \cdots b_{i+\log m - 1}$.

   Crie as cadeias

$$
\begin{aligned}
\bar{B}_1 &= \bar{b}_1 \bar{b}_{\log m + 1} \bar{b}_{2 \log m + 1} \cdots \bar{b}_{m - 2 \log m + 1} \quad , \\
\bar{B}_2 &= \bar{b}_2 \bar{b}_{\log m + 2} \bar{b}_{2 \log m + 2} \cdots \bar{b}_{m - 2 \log m + 2} \quad , \\
&\vdots \\
\bar{B}_{\log m} &= \bar{b}_{\log m} \bar{b}_{2 \log m} \bar{b}_{3 \log m} \cdots \bar{b}_{m - \log m} \quad .
\end{aligned}
$$

   OBS.: O passo 2 também usa tempo paralelo $O(\log m)$ e $O(m)$ operações.

3. Para cada sufixo de $\bar{A}$ e cada prefixo de $\bar{B}_1, \bar{B}_2, \ldots, \bar{B}_{\log m}$, calcule, em paralelo, sua característica, num total de $O(m)$ operações.

4. Dadas as cadeias

$$
\begin{aligned}
&a_0 a_1 \cdots a_{\log m - 1} \quad , \\
&a_{\log m} a_{\log m + 1} \cdots a_{2 \log m - 1} \quad , \\
&\vdots \\
&a_{m - \log m} a_{m - \log m + 1} \cdots a_{m - 1} \quad .
\end{aligned}
$$

   Calcule, em paralelo, as características de todos os seus sufixos, bem como as características de todos os prefixos de $b_0 b_1 \cdots b_{\log m - 1}$.

   OBS.: O passo 4 pode ser executado em tempo paralelo $O(\log m)$, com $\frac{m}{\log m} + 1 = O(\frac{m}{\log m})$ processadores.

5. Finalmente, determine δ em paralelo.

   Novamente, sejam $j = c_1 \log m + c_2$, $1 \leq j \leq m$, $c_1 \geq 0$, $1 \leq c_2 \leq \log m$ e $i = j - 1$. Como visto anteriormente, $\delta[i] = 1$ se, e somente se, as condições (a) e (b) são satisfeitas. Mais precisamente, se, e somente se

$$\text{(a}') \quad \chi(\bar{a}_{m - c_1 \log m}\bar{a}_{m-(c_1-1)\log m} \cdots \bar{a}_{m - \log m}) = \chi(\bar{b}_{c_2}\bar{b}_{c_2+\log m} \cdots \bar{b}_{c_2+(c_1-1)\log m}) \quad \text{e}$$

$$\text{(b}') \quad \chi(a_{m-i-1}a_{m-i} \cdots a_{m-c_1 \log m - 1}) = \chi(b_0 b_1 \cdots b_{c_2-1})$$

   são satisfeitas.

   Execute então os dois testes, que tomam tempo $O(1)$ cada. Num total de $O(m)$ operações.

V.4.4 Um algoritmo ótimo para casamento de cadeias

Dados um texto T, de tamanho n, e um padrão PAT, de tamanho m, vamos particionar T em $\lceil \frac{n}{m} \rceil$ blocos de tamanho m (possivelmente o tamanho do último bloco seja menor que m). Ou seja, o k-ésimo bloco é a cadeia

$$T[(k-1)m+1, (k-1)m+2, \ldots km], \quad k < \lceil \frac{n}{m} \rceil,$$

enquanto que o $\lceil \frac{n}{m} \rceil$-ésimo bloco é a cadeia

$$T[(\lceil \frac{n}{m} \rceil - 1) \cdot m + 1, \ldots, n].$$

Não é difícil perceber que PAT ocorre em alguma posição de T se, e somente se, para algum j, o sufixo de tamanho r do j-ésimo bloco é igual ao prefixo de tamanho r de PAT e o prefixo de tamanho $m - r$ do $(j+1)$-ésimo bloco de T é igual ao sufixo de tamanho $m - r$ de PAT.

Basta resolvermos o problema de sufixo-prefixo para PAT e os blocos de T nos dois sentidos. Ou seja, cada bloco de T como A e PAT como B, e também cada bloco de T como B e PAT como A. Cada um desses pequenos problemas pode ser executado, como vimos, em tempo paralelo $O(\log m)$, com $\frac{m}{\log m}$ processadores. Como temos um total de $2\lceil \frac{n}{m} \rceil$ desses pequenos problemas, podemos executar o algoritmo de casamento de cadeias em tempo $O(\log m)$ com $O(n/\log m)$ processadores.

V.5 Um Algoritmo Ótimo de Tempo $O(\log\log m)$

Nesta seção descrevemos o algoritmo ótimo de tempo paralelo $O(\log\log m)$ desenvolvido por Galil e Breslauer, em [BG90]. Este algoritmo é quase que inteiramente baseado nos algoritmos de [V85] e [G85], e funciona de forma similar a eles.

O ganho na complexidade de tempo paralelo dá-se em função da descoberta de que o duelo, inicialmente proposto por Vishkin [V85], funciona da mesma maneira que o máximo. Em outras palavras, duelar k posições do texto (obviamente o duelo entre duas posições tem custo unitário) é equivalente a encontrar o máximo entre k elementos. Por outro lado, encontrar o máximo entre k elementos leva, segundo [SHI81], tempo paralelo $O(\log\log k)$, com $\frac{k}{\log\log k}$ processadores de uma CRCW-PRAM.

A maior parte das instruções do referido algoritmo são instruções para modelos fortes de CRCW-PRAM. No entanto, são possíveis algumas mudanças de tal forma que se tenha o mesmo custo para modelos mais fracos. Essas alterações podem ser vistas em [BG90].

O algoritmo, assim como o de [V85] e [G85], tem duas etapas principais. A primeira é a análise do padrão, descrita na subseção V.5.2, onde o mesmo é pré-processado na forma do vetor WIT. A segunda, descrita na subseção V.5.1, procura o padrão analisado no texto.

V.5.1 A análise do texto

A técnica usada para se encontrar todas as ocorrências do padrão PAT no texto T é quase que a mesma usada por Vishkin [V85], a menos de alguns detalhes quanto à divisão do texto, relacionada ao escalonamento dos processadores.

Seja $r = \min(P, \frac{m}{2} + 1)$, onde P é o tamanho do menor período de PAT. Obviamente, $r = \frac{m}{2} + 1$ no caso aperiódico e $r = P < \frac{m}{2} + 1$ no caso periódico. Enfim, a análise do texto baseia-se em $WIT[2, \dots, r]$. Além disso, pelos lemas V.7 e V.8, em cada bloco de tamanho r de T, o padrão pode ocorrer no máximo uma vez. Ou seja, dadas duas posições candidatas em um bloco de tamanho r, pode-se duelá-las em tempo paralelo $O(1)$.

Considere o texto T subdividido em blocos de tamanho r e cada um deles dividido em blocos de tamanho $\log\log r$.

O algoritmo de análise do texto tem três etapas. Na primeira cada proces-

sador executa seqüencialmente os duelos entre as $\log\log r$ posições de cada um dos $\frac{n}{\log\log r}$ blocos de T. Isto pode ser feito em tempo $\log\log r - 1 = O(\log\log m)$. Como resultado, temos no máximo uma posição candidata para cada bloco de tamanho $\log\log r$. Na segunda etapa cada grupo de $\frac{r}{\log\log r}$ processadores executa o algoritmo ótimo de duelos, de tempo $O(\log\log(\frac{r}{\log\log r})) = O(\log\log r)$ entre as $\frac{r}{\log\log r}$ posições candidatas de cada bloco de tamanho r.

Como resultado, temos no máximo uma posição candidata para cada bloco de tamanho r do texto. A terceira e última etapa depende de PAT ser ou não periódico, e é executada de maneira idêntica àquela feita por Vishkin em [V85]. Se o padrão é aperiódico, então a terceira etapa consiste em testar, símbolo por símbolo, dada uma das $O(n/r)$ posições candidatas, num total de $O(n)$ operações, já que $r = O(\frac{m}{2})$. Agora, se o padrão é periódico ($r = P$), então a terceira etapa encontra todas as ocorrências de u^2, onde u é o menor período de PAT. Depois, para cada ocorrência de u^2, testa se é seguida por uma ocorrência de v. A partir daí, faz-se a contagem das ocorrências de uv, ou seja, a determinação dos valores do vetor $LARGEST$, como na seção V.3. O custo da terceira etapa no caso periódico é de $O(n)$ operações, como visto na seção V.3. Assim, pode ser executada em tempo $O(\log\log m)$ com $O(\frac{n}{\log\log m})$ processadores de uma CRCW-PRAM.

V.5.2 A análise do padrão

O pré-processamento do padrão também funciona de forma similar àquele feito em [V85]. Novamente faz-se o pré-processamento com prefixos de PAT de tamanhos cada vez maiores.

O algoritmo tem $2\log\log m$ passos. Seja

$$k_0 = 1 \quad ,$$

$$k_i = \frac{m^{1-2^{-i}}}{\log\log m} \quad , \quad \text{para } i = 1, \ldots, \log\log m,$$

$$k_i = 2k_{i-1} \quad , \quad \text{para } i > \log\log m.$$

Em cada psso i, o algoritmo, como em [V85], pode encontrar-se em um dos dois estados: periódico e aperiódico. Dependendo da periodicidade do primeiro bloco de tamanho k_i de PAT.

No início do passo do estado aperiódico tem-se no máximo uma posição para a qual WIT ainda não foi calculado, em cada bloco de tamanho k_{i-1}. No primeiro bloco a única posição nesta situação é a posição 1 (k_i é aperiódico).

Tenta-se calcular WIT para as posições restantes do primeiro bloco de tamanho k_i com base em $PAT[1 \cdots 2k_i]$, fazendo-se todos os testes necessários. Isto tem um total de $\frac{k_i}{k_{i-1}} \cdot 2k_i = O(m/\log\log m)$ operações, ou seja, pode ser feita em tempo $O(1)$, com $O(m/\log\log m)$ processadores.

Se todas as posições do primeiro bloco de tamanho k_i tiverem seu WIT calculado, então é porque $PAT[1 \cdots 2k_i]$ é aperiódico. Neste caso, duela-se todas as $\frac{k_i}{k_{i-1}}$ posições de cada bloco de tamanho k_i que ainda não tiveram seus WIT's calculados. Isto pode ser feito em tempo $O(\log\log \frac{k_i}{k_{i-1}}) = O(\log\log m)$ com $\frac{k_i/k_{i-1}}{\log\log m}$ processadores, usando o algoritmo equivalente ao algoritmo para se encontrar o máximo. Porém, este tempo só é necessário nos dois primeiros passos, já que a partir do terceiro tem-se menos do que $(m/\log\log m)^{1/2}$ posições para as quais não se tem WIT ainda. Nestes passos restantes, essa computação pode ser feita em tempo constante com $m/\log\log m$ processadores. Pode-se agora entrar no passo $i+1$ no estado aperiódico. Caso contrário, ou seja, se houverem posições para as quais WIT não foi calculado (além da posição 1) então é porque $PAT[1 \cdots 2k_i]$ é periódico. verifica-se, então , se a periodicidade continua em $PAT[1 \cdots k_{i+1}]$. Caso positivo o algoritmo inicia o passo $i+1$ no estado periódico. Caso contrário, calcula-se o WIT para todas as posições do primeiro bloco de tamanho k_i do tipo $\alpha p + 1$ e novamente duela-se as $\frac{k_i}{k_{i-1}}$ posições de cada bloco de tamanho k_i, como acima.

No estado periódico o algoritmo verifica se a periodicidade continua no padrão $PAT[1 \cdots k_{i+1}]$. Neste caso, calcula-se $WIT[j]$, $\forall j$, $j \neq 1 \pmod P$, no primeiro bloco de tamanho k_i, da seguinte forma. Seja $\beta = \lfloor \frac{(j-1)}{P} \rfloor \cdot P$. Faça $WIT[j] = \beta + WIT[j - \beta]$. O algoritmo então inicia o passo $i+1$ no estado periódico. Caso contrário, ou seja, se a periodicidade não continuar em $PAT[1 \cdots k_{i+1}]$, tem-se, de uma só vez, o WIT para todas as posições do tipo $\alpha p + 1$ no primeiro bloco de tamanho k_{i-1}. Duela-se, então, as posições em cada bloco de tamanho k_{i-1}. O algoritmo inicia o passo $i+1$ no estado aperiódico.

O passo inicial deve estar no estado aperiódico.

Apesar de não entrarmos em detalhes na explicação acima, existem propriedades equivalentes àquelas descritas na subseção V.3.2 (k-dispersão, k-certeza e k-limitação), que precisam ser obedecidas a cada passo do algoritmo. Isto porque os WIT's calculados para as posições $2, \ldots, 2k_i$, o são em função de $PAT[1 \cdots 2k_i]$. Além disso, o passo final requer cuidados especiais, como em [V85].

Como podemos observar, a idéia básica dos dois algoritmos (o descrito nesta seção ([BG90]) e o descrito na seção V.3.([V85])) é a mesma. a diferença está no tamanho do prefixo, a cada passo, que orienta as instruções, para cada estado.

Além disso, pelas considerações feitas no capítulo IV, o algoritmo apresentado nesta seção tem complexidade de tempo ótima, já que usa tempo paralelo $O(\log\log m)$.

V.6 Casamento de Cadeias com Pré-processamento Lento

Em [V90] é apresentado um algoritmo paralelo para casamento de cadeias que usa tempo paralelo $O(\log^* n)$. É de se estranhar tal complexidade depois que se lê o capítulo IV. Por outro lado, como dissemos anteriormente, os limites inferiores apresentados no capítulo IV se referem ao problema de casamento de cadeias incluindo o tempo paralelo gasto em qualquer espécie de pré-processamento do padrão . Já o algoritmo devido a Vishkin [V90], apresentado nesta seção , necessita, para a obtenção de tempo paralelo $O(\log^* n)$ na análise do texto, de um pré-processamento bastante lento. Precisamente $O(\frac{\log^2 m}{\log\log m})$.

O algoritmo usa uma nova técnica bastante interessante para a análise do padrão. Consiste em determinar um conjunto de no máximo $\log m$ posições do padrão (de tamanho m), de tal forma que, encontrado tal conjunto no texto, é possível desqualificar algumas posições vizinhas sem qualquer relevante custo adicional, caso o padrão seja aperiódico. Assim, consegue-se uma busca mais rápida do padrão no texto.

Tal conjunto de posições é chamado de *deterministic sample* pelo autor. Nós o chamaremos de *assinatura* do padrão .

Na subseção V.6.1 introduziremos a noção de assinatura, bem como as idéias necessárias para o entendimento do algoritmo. Na subseção seguinte mostraremos como é feita a análise do padrão. Na subseção V.6.3 descreveremos duas análises para o texto, a partir das quais é desenvolvida a análise com a complexidade desejada, descrita na subseção V.6.4. Na subseção V.6.5 é tratado o caso periódico.

V.6.1 A assinatura

A amostra determinística do padrão, a qual chamamos de assinatura, consiste de uma seqüência ordenada $A = [A(1), \cdots, A(l)]$, onde cada $A(j)$, $1 \leq j \leq l$, é uma posição diferente do padrão, ou seja, $A(j)$ é um inteiro entre 1 e m. Além disso, $l \leq \log m - 1$.

A idéia básica inicial consiste em testar, para cada uma das $n - m + 1$ posições do texto candidatas a uma ocorrência do padrão, se as l posições da assinatura emparelham com as do texto. A surpreendente revelação é que, a partir da descoberta de uma posição candidata r a partir da qual a assinatura

ocorre, pode-se, em um passo paralelo, eliminar várias posições de tal forma que apenas uma sobreviva num bloco de tamanho $\frac{m}{2}$ do texto.

Os conceitos de periodicidade, bem como o vetor WIT são considerados também em [V90].

V.6.2 O pré-processamento do padrão

A determinação de uma assinatura do padrão PAT é o seu pré-processamento. Convenientemente consideraremos apenas o caso em que o padrão é aperiódico. No final desta seção falaremos como se resolve o problema no caso periódico (lembre-se que, mesmo que o padrão seja periódico, com menor período de tamanho P, seu prefixo de tamanho $2P-1$ é aperiódico).

O pré-processamento consiste de três passos. O primeiro passo nada mais é do que a construção do vetor WIT. Pode ser feito utilizando-se, por exemplo, o algoritmo de [V85], de tempo paralelo $O(\log m)$, com número ótimo de processadores. O vetor WIT será necessário somente no pré-processamento do padrão e pode ser desprezado antes mesmo do início da análise do texto.

Vamos supor, sem perda de generalidade, que m é par e considerar $\frac{m}{2}$ cópias $c_1, c_2, \cdots, c_{m/2}$ de PAT dispostas como na figura V.4, de tal modo que a primeira posição da cópia c_i pertence a mesma coluna da i-ésima posição de c_1, e isto acontece sucessivamente com as posições subseqüentes de c_1 e c_i, $2 \leq i \leq \frac{m}{2}$.

Figura V.4

A construção da assinatura A dependerá diretamente da determinação da seqüência $\bar{A}$ de posições descrito abaixo.

Selecione no máximo $\log m - 1$ colunas $\bar{A}(1), \bar{A}(2), \ldots, \bar{A}(l)$ e a cada coluna $\bar{A}(j)$ associe um símbolo $s(\bar{A}(j))$, $1 \leq j \leq l$, tal que exista exatamente uma cópia c_x para a qual:

(i) c_x intersecta todas essas l colunas, ou seja, $x \leq \bar{A}(j) < x+m, \forall 1 \leq j \leq l$;

(ii) para cada coluna $\bar{A}(j)$, o símbolo em c_x é igual ao símbolo associado a ela, ou seja, $PAT[\bar{A}(j) - x + 1] = s(\bar{A}(j))$, para todo j, $1 \leq j \leq l$.

O passo 2 do pré-processamento consiste na determinação de tais colunas. O passo 3 determina diretamente, a partir das colunas do passo 2, uma assinatura para o padrão PAT.

A própria execução do passo 2, que é indutiva, prova a existência de tal conjunto de colunas. O passo 2 constói, em no máximo l etapas, a sequência $\bar{A}$, da seguinte forma.

Na etapa inicial tem-se o conjunto $I_0 = \{c_1, \cdots, c_{\frac{m}{2}}\}$. Para cada i, $0 \leq i \leq l$, $|I_{i+1}| \leq \frac{|I_i|}{2}$. Além disso, $|I_l| = 1$. No início, c_1 e $c_{\frac{m}{2}}$ são ditos serem, respectivamente, a cópia mais a esquerda e mais a direita em I_0.

Na etapa indutiva, se I_i tem exatamente um elemento, então $l = i$ e está terminado o passo 2. Caso contrário, constrói-se um subconjunto I_{i+1} não vazio de I_i, tal que $|I_{i+1}| \leq \frac{|I_i|}{2}$. Isto pode ser feito da seguinte forma. Sejam c_E e c_D as cópias mais a esquerda e mais a direita, respectivamente, de I_i. Como PAT é aperiódico, WIT fornecerá uma coluna $\bar{A}(i+1)$ que conterá dois símbolos diferentes para as cópias c_E e c_D. Note que $\bar{A}(i+1)$ intersecta todas as colunas de I_i. Para cada um desses dois símbolos, encontre quantos são iguais a um e quantos são iguais ao outro. Associe a esta coluna $\bar{A}(i+1)$ aquele símbolo que menos aparece em $\bar{A}(i+1)$ entre os dois, nas cópias de I_i. Note que, neste caso, o símbolo associado aparece no máximo $\frac{I_i}{2}$ vezes. Faça I_{i+1} ser igual ao conjunto de cópias de I_i que tem esse símbolo na coluna $\bar{A}(i+1)$.

Cada etapa, portanto, escolhe uma coluna para a sequência $\bar{A}$. No pior caso, temos $l = \log m - 1$ etapas.

Podemos supor agora que as cópias de I_i já estejam num vetor compactado ao entrarmos na etapa $i+1$. Isto pode ser feito usando soma de prefixos. A etapa $i+1$ toma tempo paralelo $O(\frac{\log |I_i|}{\log\log |I_i|})$ e tem $O(|I_i|)$ operações . $|I_i|$ decresce geometricamente. O passo 2 leva, então, tempo $O(\frac{\log^2 m}{\log\log m})$ e tem um total de $O(m)$ operações .

O passo 3 se resume em apenas "transladar" as colunas escolhidas no passo 2, de tal forma que $A = [A(1), \cdots, A(l)] := [\bar{A}(1) - x + 1, \cdots, \bar{A}(l) - x + 1]$, onde x é a única cópia de PAT pertencente ao I_l.

Todo o pré-processamento requer tempo $O(\frac{\log^2 m}{\log \log m})$ e $O(m)$ operações , já que o passo 2 domina toda a sua complexidade. Escritas concorrentes acontecem na determinação de WIT, bem como na soma de prefixos do passo 2.

V.6.3 Duas análises não-ótimas para o texto

Nesta subseção são apresentados dois algoritmos para a análise do texto T. Em outras palavras, dois algoritmos que procuram PAT em T, com base na assinatura de PAT, determinada pelo algoritmo visto na subseção anterior.

Estes dois algoritmos, respectivamente chamados de algoritmo 1 e algoritmo 2 são tais que o primeiro é de tempo paralelo constante, mas necessita de $O(n \log m)$ operações. O segundo precisa de tempo paralelo $O(\log m)$ e $O(n)$ operações. Na subseção V.6.5 combina-se estes dois algoritmos para a obtenção de um terceiro, este de tempo paralelo $O(\log^* n)$.

Para qualquer desses algoritmos de análise do texto supõe-se que o padrão é aperiódico. Na próxima subseção mostramos como o caso periódico pode ser resolvido.

Suponhamos que o prefixo de T, de tamanho $n' = n - m + 1$ esteja particionado em sucessivas cadeias de $\frac{m}{2}$ símbolos cada (possivelmente a última com menos símbolos).

O algoritmo 1 tem três passos básicos. No primeiro passo, para cada posição i de T, $1 \leq i \leq n'$, checa se as seguintes l igualdades valem:

$$T[i-1+A(j)] =^? PAT[A(j)], \qquad \forall A(j) \in A \quad (1 \leq j \leq l)$$

Em outras palavras, testa se a assinatura A de PAT ocorre a partir da posição i do texto T.

O passo 1 tem um total de $O(n \log m)$ operações, sendo $O(\log m)$ para cada uma das n' posições inicialmente candidatas.

O passo 2 precisa de uma propriedade interessante a respeito da assinatura do padrão. Seja c_x a única cópia resultante do passo 3 do algoritmo de pré-processamento de PAT. Ou seja, c_x é a única cópia que emparelha com as posições da assinatura A. Seja i uma posição candidata de T que já tenha

sobrevivido ao teste do passo 1. Então pode-se eliminar as $x-1$ posições precedentes a i, bem como as $\frac{m}{2}-x$ posições que sucedem i. Ou seja, as posições $i-x+1,\ldots i-1, i+1,\ldots, i+\frac{m}{2}-x$ não podem mais ser candidatas.

Isto significa que numa cadeia de $\frac{m}{2}$ símbolos sucessivos de T, não pode haver mais do que duas posições candidatas. Então o passo 2 encontra, para cada bloco de tamanho $\frac{m}{2}$ de T, as posições candidatas mais a esquerda e a mais a direita resultantes do passo 1. Depois elimina qualquer outra posição candidata neste bloco. Como resultado, no máximo duas posições candidatas sobrevivem em cada bloco de tamanho $\frac{m}{2}$.

O passo 3 se resume em testar, símbolo por símbolo, todas as m posições de PAT para cada posição candidata sobrevivente de T.

O passo 1 do algoritmo 1, como dissemos anteriormente, tem custo $O(n \log m)$. No passo 2, a determinação das posições candidatas extremas de cada bloco de tamanho $\frac{m}{2}$ de T pode ser feita em tempo paralelo $O(1)$ com $\frac{m}{2}$ processadores de uma CRCW-PRAM-Fraca. A eliminação das posições não extremas tem a mesma complexidade. O passo 3 tem um total de $2m\lceil\frac{n-m+1}{m/2}\rceil = O(n)$ operações de teste.

O algoritmo 1 usa, então, tempo paralelo $O(1)$ com $O(n \log m)$ processadores, não sendo, portanto, ótimo.

Uma alternativa para o passo 1 do algoritmo 1 seria executá-lo em $O(\log m)$ etapas. Em cada etapa j se testaria a j-ésima posição da assinatura de PAT com a $A(j)$-ésima posição de PAT. Mesmo assim o número total de operações de teste continuaria sendo $O(n \log m)$.

O algoritmo 2 descrito a seguir usa a idéia alternativa acima e, além disso, a cada etapa j, elimina algumas posições candidatas de acordo com a propriedade vista a pouco.

O algoritmo 2 tem $l+1$ passos, cada passo tem 3 etapas. O primeiro passo, em particular, é diferente dos outros. Nós o explicaremos primeiro, em separado.

As três etapas do primeiro passo são as seguintes: primeiro testa-se, para cada uma das n' posições candidatas i do texto, se vale a seguinte igualdade

$$T[i+A(1)-1] =^{?} PAT[A(1)].$$

Depois, para cada bloco de tamanho $\frac{m}{2}$ do texto T, são executadas, em paralelo, as duas outras etapas, quais sejam: Encontra-se as posições candidatas a e b de T, respectivamente, a mais a esquerda e a mais a direita do bloco, ou seja, a é a posição mais a esquerda tal que $T[a+A(1)-1] = PAT[A(1)]$ e b é a posição mais a direita tal que $T[b+A(1)-1] = PAT[A(1)]$. Sejam $E = a+A(1)-1$ e $D = b+A(1)-1$. Obviamente não se pode afirmar que a e b sejam as únicas posições candidatas de fato neste bloco. Por outro lado, serão candidatas neste bloco aquelas posições k tais que $PAT[E-k+1] = T[E]$ ou $PAT[D-k+1] = T[D]$. Imagine agora o conjunto I_1, de cópias de PAT, resultante da análise do padrão ,

disposto sobre o texto (disposto da mesma forma que na figura V.4), de tal modo que a coluna $\bar{A}(1)$ esteja na posição E (o mesmo vale para D).

Então o conjunto I_1 por si só nos dirá quais as posições candidatas neste bloco. São as posições que estão "embaixo" da primeira posição em cada cópia de I_1. Temos então um conjunto T_E de posições correspondentes a E e um conjunto T_D de posições correspondentes a D.

Concluindo, uma posição será candidata se pertencer a T_E ou a T_D. Note que $a \in T_E$ e $b \in T_D$. Não é difícil nos certificarmos de tal propriedade.

A última etapa do passo 1 usa, então, o conjunto I_1 para construir T_E e T_D, para cada bloco de tamanho $\frac{m}{2}$ de T, como descrito acima.

Vamos agora descrever o passo α, $2 \leq \alpha \leq l$. Novamente cada passo α é executado em paralelo, para cada bloco de tamanho $\frac{m}{2}$ do texto. Vamos descrever o passo α para um único bloco.

No decorrer dos passos α podem surgir algumas posições falsas, a elas chamaremos de *candidatas oficiosas.* As outras serão as *candidatas oficiais* (mais tarde diremos como podem surgir tais candidatas oficiosas).

O passo α, $2 \leq \alpha \leq l$ tem três etapas, similares as do passo 1. A primeira etapa considera as posições candidatas oficiais resultantes das listas T_E e T_D do passo $\alpha - 1$. Para cada tal candidata i, verifica a igualdade abaixo:

$$T[i + A(\alpha) - 1] =^? PAT[A(\alpha)].$$

Ou seja, testa se a α-ésima posição da assinatura A de PAT é igual a $A(\alpha)$-ésima posição de PAT.

A segunda etapa encontra, entre as resultantes da primeira etapa, as posições mais a esquerda a e mais a direita b no bloco. Ou seja, $a(b)$ é a posição mais a esquerda(direita) no bloco tal que

$$t[a + A(\alpha) - 1] = PAT[A(\alpha)]$$

$(T[b + A(\alpha) - 1] = PAT[A(\alpha)])$. Novamente, sejam $E = a + A(\alpha) - 1$ e $D = b + A(\alpha) - 1$. A terceira e última etapa do passo α constrói os conjuntos T_E e T_D como no passo 1, só que agora usando a coluna $\bar{A}(\alpha)$ como âncora.

Resta agora o último dos $l + 1$ passos do algoritmo 2, que checa, símbolo por símbolo, se nas posições candidatas restantes de fato ocorre PAT.

Uma posição candidata oficiosa pode surgir por uma das razões :

(1) Ela não pertencia nem a T_E nem a T_D, no passo $\alpha - 1$;

(2) Ela já era candidata oficiosa no passo $\alpha - 1$ e satisfez a igualdade para a α-ésima posição da assinatura A;

(3) Ela era uma candidata oficial no passo $\alpha - 1$, mas falhou no teste da α-ésima posição da assinatura A.

O tempo paralelo gasto em cada passo α, $1 \leq \alpha \leq l$ é o mesmo e é constante. Como os índices dos conjuntos T_E e T_D, que contém elementos de $I_{\alpha-1}$, já foram calculados no pré-processamento de PAT (usando soma de prefixos), é direto referenciá-los. Por outro lado, encontrar as posições candidatas mais a esquerda e mais a direita também tem tempo paralelo $O(1)$. Além disso, o número de elementos de T_E e T_D é limitado por $\frac{m}{2^{\alpha+1}}$, no passo $\alpha+1$, que decresce geometricamente (à razão de 2) no decorrer do algoritmo. No último passo, temos $O(n/m)$ posições candidatas e m operações por posição. Logo o número total de operações do algoritmo 2 é dado por

$$O(\sum_{j=1}^{\log m-1} \frac{m}{2^j}) \cdot O(n/m) = O(n)$$

operações .

V.6.4 A análise de tempo $O(\log^* n)$

A função $\log^* x$ é definida como segue. Seja $\log^{(1)} x = \log x$. Seja também $\log^{(i)} x = \log \log^{(i-1)} x$. Então

$$\log^* x = \min_i \{\log^{(i)} x \leq 2\}.$$

O algoritmo descrito a seguir executa a análise do texto em tempo $O(\log^* n)$ com um total de $O(n)$ operações . Tem três etapas. Na primeira etapa executa os $\delta = 2 \log \log^* n + 2$ passos do algoritmo 2, descrito anteriormente. Ou seja, utiliza os primeiros δ símbolos da assinatura A de PAT. Como resultado, temos um total, entre todos os T_E 's eT_D ', de

$$\frac{n}{m/2} \cdot 2 \cdot \frac{m}{2^{2 \log \log^* n+3}} = O(\frac{n}{(\log^* n)^2})$$

posições candidatas (oficiais ou não).

A segunda etapa tem $\log^* n$ iterações. Para cada iteração, as entradas são os elementos dos conjuntos T_E e T_D, para todos os blocos de tamanho $\frac{m}{2}$ de T. Como o número de posições candidatas vai diminuindo a medida em que as iterações vão sendo executadas, a cada iteração faz-se mais testes por cada posição candidata. O conjunto de instruções abaixo descreve esta etapa.

para $c := \log^* n$ **até** 1 **faça**

início

{ *entrada para essa iteração: no máximo* $\frac{n}{\log^{(c)} n \log^* n}$ *candidatas* }

para cada candidata oficial **faça**

teste as próximas $\log^{(c)}$ posições da assinatura A ;

para cada bloco de tamanho $\frac{m}{2}$ **faça**

início

encontre as posições candidatas oficiais mais a esquerda e mais a direita ;

construa T_E e T_D

fim

fim

Note que na primeira iteração serão testadas as posições

$$A(\delta + 1), A(\delta + 2), \ldots, A(\min\{l, \delta + \log^{(c)} n)$$

de A. Note ainda que a segunda etapa terminará tendo toda a assinatura testada. Além disso, são feitos $O(\log^{(c)} n)$ testes por iteração , por posição candidata. Como há no máximo $\frac{n}{\log^{(c)} n \log^* n}$ posições candidatas por iteração, o número total de operações em cada iteração é $O(n/\log^* n)$. Temos entaão um total de $O(n)$ operações nas $O(\log^* n)$ iterações .

A última etapa do algoritmo novamente faz, símbolo por símbolo, os m testes para cada uma das duas posições candidatas de cada bloco (um total de $O(n/m)$ candidatas). Temos assim uma total de $O(n)$ operações em tempo $O(\log^* n)$ com um número ótimo de processadores.

V.6.5 O caso periódico

Como dissemos anteriormente, assumimos, desde o início, que PAT é não periódico. Suponhamos agora que PAT seja periódico tal que $PAT = u^k v$, onde u, de tamanho P, é o menor período de PAT e v é um prefixo de u, $k > 1$. Suponhamos ainda que todas as ocorrências de $PAT[1 \cdots 2p-1]$, que é aperiódico, tenham sido encontradas pelo algoritmo da subseção anterior.

Como sabemos (pelo caso periódico da seção V.3) temos no máximo $O(n/P)$ ocorrências de u^2.

Usa-se, então, a mesma técnica utilizada no caso periódico do algoritmo de Vishkin [V85], da seção V.3, cujo número total de operações é $O(n)$.

V.7 Um Algoritmo de Tempo Constante

Nesta seção descrevemos o algoritmo de tempo paralelo $O(1)$ e $O(n)$ processadores, proposto por Galil, em [G92]. Tal algoritmo tem como base a idéia de que, ao invés de se procurar todo o padrão no texto, talvez seja mais interessante procurar partes convenientemente menores dele. A princípio parece semelhante ao algoritmo da seção anterior [V90]. Mas não é.

O algoritmo necessita de um pré-processamento caro do padrão. Logo, não se inclui na classe dos algoritmos do tipo que o custo do pré-processamento é incluído (capítulo 4).

A seguir, descrevemos o algoritmo.

Primeiramente, vamos dividir o texto T em sub-textos sobrepostos de tamanho $\frac{5m}{4}$, do tipo

$$T_j = T[(j-1)\frac{m}{4}+1, \cdots, (j+4)\frac{m}{4}].$$

Cada grupo de $m/4$ processadores ficará responsável por encontrar todas as ocorrências de PAT no primeiro quarto de T_j, $1 \leq j \leq x$, onde é o menor inteiro maior ou igual a $\frac{4n}{m} - 4$ (possivelmente o último sub-texto tenha menos que $\frac{5m}{4}$

posições). Cada sub-texto T_j é manuseado em paralelo com $\frac{m}{4}$ processadores. No total, serão necessários $\frac{m}{4} \cdot O(\frac{n}{m/4}) = O(n)$ processadores.

Além disso, podemos (e vamos precisar) supor, sem perda de generalidade, que PAT é aperiódico. O caso oposto pode ser resolvido como em [V85] e [BG90]. Procura-se no texto o padrão $PAT[1 \cdots 2P-1]$, onde P é o tamanho do menor período de PAT. Essa computação requer um total de $O(n)$ operações.

Vamos nos concentrar agora em apenas um dos T_j e supor que PAT ocorre em T. Temos então um padrão PAT de tamanho m e um texto de tamanho $\frac{5}{4}m$. Lembre-se que temos disponíveis apenas $O(\frac{m}{4})$ processadores para procurarmos PAT em T_j.

Para simplificar a notação, como

$$T_j = T[(j-1)\frac{m}{4} + 1, \cdots (j+4)\frac{m}{4}],$$

trabalharemos com

$$T_j[1 \cdots \frac{5}{4}m].$$

Seja w uma subcadeia de PAT. Como parte da ocorrência de PAT no texto, w também ocorre no texto. Chamamos estas duas ocorrências de *ocorrências duais*.

Suponhamos que nos seja dado uma sub-cadeia z, de tamanho $\frac{\log m}{4}$ no segundo quarto de PAT. Suponhamos ainda que tenhamos encontrado, em tempo $O(1)$, com $O(\frac{m}{4})$ processadores, uma ocorrência de z no segundo ou terceiro quarto de T_j. Supondo que PAT ocorre em T_j, certamente z satisfaz as seguintes propriedades:

(i) z ocorre no segundo ou terceiro quarto de T_j;

(ii) cada ocorrência de z no segundo ou terceiro quarto do texto tem uma ocorrência dual num dos primeiros três quartos de T_j.

Se z é aperiódico, então, pelo lema V.7, z ocorre no máximo uma vez em cada bloco de tamanho $\frac{\log m}{8}$. Na análise do padrão são encontradas todas as ocorrências de z nos primeiros 3 quartos de PAT. Essas ocorrências são armazenadas nos primeiros $O(\frac{m}{\log m})$ processadores (pois z pode no máximo $\frac{m/4}{(\log m)/8} \cdot 3$ nos três primeiros quartos de PAT). Uma dessas ocorrências deve ser a dual de z encontrado em T_j.

Seja r a posição de ocorrência de z encontrada em T_j. Cada posição dual de z subtraída de r é uma posição candidata a ocorrência de PAT em T_j.

Logo, temos no máximo uma posição candidata em cada bloco de tamanho $\frac{\log m}{8}$ no primeiro quarto de T_j. Que nos leva a um total de $O(\frac{m}{\log m})$ posições candidatas no primeiro quarto de T_j. Para cada uma dessas posições , faz-se o teste

da assinatura de PAT, descrito na seção V.6, num total de $O(\frac{m}{\log m}) \cdot O(\log m)$ operações. Ou seja, em tempo $O(1)$ com $O(m)$ processadores. Com isso, pelas propriedades da assinatura de PAT, temos no máximo uma posição candidata no primeiro quarto de T_j. Como resultado, temos um total de $O(\frac{n}{m/4})$ posições candidatas em T. Faz-se, então, o teste, símbolo por símbolo, de todas essas posições em tempo $O(1)$ com $O(n)$ processadores.

Agora, se z é periódico, então são necessários novos conceitos, dados abaixo.

Definição V.3 – Dado z, uma sub-cadeia periódica de w, um *segmento* é uma sub-cadeia maximal de w contendo z tendo menor período de mesmo tamanho que o de z. O primeiro (último) z do segmento é a ocorrência mais a esquerda (direita) de z em w. ▮

Lema V.11 – Dados uma ocorrência de z, periódico com menor período de tamanho P, pode-se encontrar o primeiro e o último z em w, em tempo $O(1)$ e $O(|w|)$ processadores.

Prova – Seja j a posição dada como ocorrência de z em w. Sejam k a posição de ocorrência do segmento e r a posição de ocorrência do primeiro z do segmento. Seja o vetor $S[1 \cdots |w|]$ tal que $S[0] = 0$ e $S[i] = 1$, se $w[i] = w[i + P]$, e $S[i] = 0$ caso contrário. k é o maior inteiro menor que j com $A[k-1] = 0$. r é o menor inteiro maior ou igual a k tal que $r \equiv j \pmod{P}$. Ambos, k e r, podem ser encontrados em tempo $O(1)$, já que o mínimo e o máximo entre 1 e $|w|$ o podem. O último z pode ser encontrado de forma análoga. ▮

Voltamos então ao caso em que z é periódico. Seja α o segmento de z encontrado em T_j e β o segmento no padrão correspondente a α. α e β não necessariamente se emparelham, já que α pode continuar além da ocorrência do padrão. O lema abaixo resolve este problema.

Lema V.12 – Ou o primeiro z de α é dual ao primeiro z de β, ou o último z de α é dual ao último z de β.

Prova – Como PAT é aperiódico, ou β começa depois do início de PAT, ou termina antes de seu final. No primeiro caso, o primeiro z de α é dual ao primeiro z de β. No segundo caso o último z de α é dual ao último z de β. ▮

Na análise do padrão todos os segmentos de PAT referentes a z são encontrados. Além disso, para cada segmento encontrado, busca-se seu primeiro e seu último z. Dois segmentos não podem se sobrepor em mais do que $\frac{|z|}{2} - 1$ posições, pois caso contrário não seriam maximais. Então no máximo um primeiro (último) z pode ocorrer em cada bloco de tamanho $\frac{|z|}{2} = \frac{\log m}{8}$.

Resumindo, se z é periódico, então encontra-se o primeiro e o último z de seu segmento (α). Isto pode ser feito em tempo $O(1)$ com $O(m/4)$ processadores. Como no caso anterior (aperiódico), encontra-se posições candidatas através do primeiro z de α e os primeiros z's dos segmentos de PAT. O mesmo é feito com relação ao último z de β e os últimos z's dos segmentos de PAT. Temos no máximo uma posição candidata a ocorrência de PAT em cada bloco de tamanho $\frac{\log m}{8}$ no primeiro quarto de T_j.

As posições candidatas são testadas de forma análoga ao caso de z ser aperiódico, num total de $O(n)$ operações.

Nosso problema se resume então em escolher convenientemente a sub-cadeia z no segundo quarto de PAT e encontrá-la, em tempo paralelo $O(1)$, no segundo ou no terceiro quarto de T_j.

Com este propósito, Galil divide este problema em três casos, em função do tamanho do alfabeto Σ' dos símbolos encontrados no segundo quarto de PAT.

No primeiro caso, em que $|\Sigma'| = 2$, seja z a sub-cadeia de tamanho $\frac{\log m}{4}$ mais frequente no segundo quarto do padrão (o pré-processamento trata de resolver isso). Ela ocorre no mínimo

$$\frac{m/4}{2^{\frac{\log m}{4}}} = \frac{m}{4 \cdot m^{0.25}} > m^{0.7}$$

vezes no segundo ou terceiro quarto de T_j. Encontra-se uma tal ocorrência de z da seguinte forma. Constrói-se um conjunto H, de posições do segundo e terceiro quartos de T_j, então pelo menos uma posição de H é uma ocorrência de z. Esse conjunto é construído na análise do padrão utilizando-se o método guloso.

Seja M uma matriz de $\frac{m}{4}$ linhas $1, \ldots, \frac{m}{4}$ e $\frac{m}{2}$ colunas $\frac{m}{4}+1, \ldots, 3\frac{m}{4}$. As linhas se referem ao primeiro quarto de T_j, enquanto que as colunas se referem aos segundo e terceiro quartos.

A matriz M é da seguinte forma. Se z ocorre nas posições $j_1, j_2, \cdots$, então $M(i, i+j_1-1), M(i, i+j_2-1), \ldots$ são iguais a 1, $1 \leq i \leq \frac{m}{4}$. As demais posições são iguais a zero. Em outras palavras, M tem um 1 para a dual de cada ocorrência de z no segundo quarto de PAT.

O algoritmo tem k passos. A cada passo, escolhe uma coluna. H terá, então, k elementos. especificamente, o algoritmo escolhe a coluna que tem maior número de 1's. Em seguida elimina todas as linhas que tem 1 na coluna escolhida.

Seja t_i o número de 1's não eliminados após a escolha de i colunas, ou seja, após o passo i. A $(i+1)$-ésima coluna a ser escolhida certamente terá no mínimo

$$\frac{t_i}{\frac{m}{2} - i} \geq \frac{t_i}{m} 1\text{'s}.$$

Logo,

$$t_{i+1} \leq t_i - \frac{t_i}{m} \cdot m^{0.7},$$

já que a eliminação de uma linha acarreta na eliminação de pelo menos $m^{0.7}$ 1's. Como $t_0 < m^2$ e $t_{i+1} \leq t_i(1 - \frac{1}{m^{0.3}})$, temos que

$$t_i \leq t_0(1 - \frac{1}{m^{0.3}})^i < m^2(1 - \frac{1}{m^{0.3}})^i, \quad i = 1, 2, \ldots.$$

Vamos mostrar que após $k = m^{0.3} \log m^2$ passos, temos $t_k < 1$. Além disso, $k = O(\frac{m}{\log m})$.

Lema V.13 – $k = m^{0.3} \log m^2 = O(\frac{m}{\log m})$.
Prova – Devemos mostrar que $\exists$ inteiros c, m_0, tais que

$$m^{0.3} \log m^2 \leq c \frac{m}{\log m}, \quad \forall m \geq m_0.$$

Ou seja, mostrar que

$$\frac{2 \log^2 m}{m^{0.7}} \leq c, \quad \forall m \geq m_0.$$

Sabemos que

$$\lim_{m \to \infty} \frac{2 \log^2 m}{m^{0.7}} = 0.$$

Em outras palavras, dado $\epsilon = 1$, $\exists m_1$ tal que

$$\frac{2 \log^2 m}{m^{0.7}} \leq 1, \quad \forall m \geq m_1.$$

Tome $c = \epsilon = 1$ e $m_0 = m_1$. Então

$$\frac{2 \log^2 m}{m^{0.7}} \leq 1, \quad \forall m \geq m_0.$$

Portanto, $k = O(\frac{m}{\log m})$. ∎

Lema V.14 – $t_k < 1$.
Prova – Sabemos que

$$t_k < m^2(1 - \frac{1}{m^{0.3}})^{m^{0.3} \log m^2}.$$

Vamos supor, por absurdo, que

$$m^2(1 - \frac{1}{m^{0.3}})^{m^{0.3} \log m^2} \geq 1.$$

Assim,

$$(1 - \frac{1}{m^{0.3}})^{m^{0.3} \log m^2} \geq \frac{1}{m^2}$$
$$m^{0.3} \log m^2 \log(1 - \frac{1}{m^{0.3}}) \geq -\log m^2$$
$$\log(1 - \frac{1}{m^{0.3}}) \geq -\frac{1}{m^{0.3}}.$$

Absurdo, pois $0 < \frac{1}{m^{0.3}} < 1$ e $\log(1-x) < -x$, $\forall x$, $0 < x < 1$. Portanto, $t_k < 1$. ∎

Resumindo, como $k = O(\frac{m}{\log m})$, cada um dos k primeiros processadores responsáveis pelo primeiro quarto de T_j terá uma posição de H e $\frac{\log m}{4}$ processadores para testar uma ocorrência de z, em tempo $O(1)$, num total de $O(m/4)$ processadores.

Se PAT ocorre no primeiro quarto de T_j, então algum z será encontrado. Pode ser que haja alguma ocorrência de z e PAT não ocorra no primeiro quarto de T_j. Isto não é um problema, pois tal posição será eliminada mais tarde. Agora, se z não é encontrado, é porque PAT não ocorre no primeiro quarto de T_j e então todas as posições deste quarto são eliminadas.

No segundo caso, onde $|\Sigma'| > \log m$, existe um símbolo no segundo quarto de PAT que aparece menos do que $\frac{m}{\log m}$ vezes em PAT. No pré-processamento encontramos este símbolo e todas as suas ocorrências no padrão, e as armazenamos nos primeiros $\frac{m}{\log m}$ processadores. Se este símbolo ocorre em T_j, então satisfaz as duas propriedades (i) e (ii). Em tempo $O(1)$, com $\frac{m}{2}$ processadores, encontramos uma ocorrência deste símbolo no segundo ou terceiro quarto de T_j. Como resultado temos no máximo $O(m \log m)$ posições candidatas no primeiro quarto de T_j. Novamente, faz-se os testes da assinatura de PAT ($O(\log m)$ operações) para cada uma das $O(\frac{m}{\log m})$ posições candidatas. Restará apenas uma posição candidata no primeiro quarto de T_j.

No terceiro e último caso, onde $2 < |\Sigma'| \leq \log m$ a solução é similar a do primeiro caso. Só que agora $|z| = \log\log m$. O número de sub-cadeias distintas de tamanho $\log\log m$ é $(\log m)^{\log\log m} < m^{\epsilon}$, então existe uma que repete mais do que $\frac{m}{m^{\epsilon}} > m^{0.7}$ vezes. Logo, o tamanho de H novamaente será pequemo.

A análise do padrão consiste, além do pré-processamento de Vishkin [V90], descrito na seção anterior, de algumas tarefas caras, tais como: escolher a mais frequente sub-cadeia z de tamanho $\frac{\log m}{4}$, no caso em que $|\Sigma|' = 2$; encontrar todas as ocorrências de z no três primeiros quartos de PAT; encontrar todos os segmentos de PAT referentes a z; construir o conjunto H; calcular $|\Sigma'|$ e possivelmente as frequências dos símbolos de Σ'.

Segundo o autor todo o pré-processamento adicional pode ser feito em tempo $O(m^2)$. Além disso, propõe que se faça algumas suposições, como por exemplo, a de que o tamanho do padrão seja $O(\log\log m)$. Assim, o custo de seu pré-processamento não ficaria mais caro que o de Vishkin.

Convém ressaltarmos que este custo possa talvez ser ainda melhorado. O autor deixa esta questão em aberto. Além disso, se trata de um algoritmo que necessita de um pré-procesamento lento. Portanto, não se inclui na classe dos algoritmos ótimos, no que diz respeito ao limites inferiores encontrados em [BG91], descritos no capítulo IV.

Capítulo VI

Um Algoritmo Para o Modelo CREW-PRAM

Como vimos na seção V.3, o algoritmo de Vishkin [V85] usa tempo paralelo $O(\log m)$ com um número ótimo de processadores de uma CRCW-PRAM. Além disso, vimos que, apesar de sua análise do texto poder ser executada com o mesmo custo numa CREW-PRAM, sua análise do padrão requer tempo paralelo $O(\log^2 m)$ nessa máquina.

Neste capítulo propomos um algoritmo para a análise do padrão, ou seja, um algoritmo para o cálculo do vetor WIT, de tempo paralelo $O(\log m)$, com $O(\frac{n}{\log m})$ processadores de uma CREW-PRAM, que funciona para os casos em que o tamanho do alfabeto é fixo e $m = O(\log^2 n)$.

Primeiramente vamos descrever o que chamamos de algoritmo 1, que foi desenvolvido para o caso geral (sem qualquer restrição sobre o alfabeto ou sobre o padrão). Tal algoritmo não é ótimo no sentido de que manipula (escreve e lê) valores numéricos na memória da PRAM, para os quais o custo dessas operações é $O(m)$, e não $O(1)$ como se espera. Mais tarde veremos por quê isto acontece.

O algoritmo 1 se baseia no fato de que, ao supormos, sem perda de generalidade, que o alfabeto $\Sigma \subseteq \{0, 1, \ldots, n-1\}$, podemos ver PAT como sendo um número inteiro, de m dígitos, escrito na base n. Ou seja, podemos ver $PAT[1 \cdots m]$ como sendo o número dado por

$$\sum_{i=0}^{m-1} PAT[m-i] \cdot n^i.$$

É claro que estamos levando em conta o fato de que um número inteiro não-negativo é escrito de forma única numa base qualquer.

Seguindo esta idéia, o cálculo de $WIT[j]$, $1 \leq j \leq m$, que se constitui na comparação do padrões $PAT[1 \cdots m-j+1]$ e $PAT[j \cdots m]$, se reduz à comparação entre o valor de

$$A_j = \sum_{i=m-(m-j+1)}^{m-1} PAT[m-i] \cdot n^{i-j+1} \quad ,$$

que é o valor numérico de $PAT[1 \cdots m-j+1]$, e o valor de

$$B_j = \sum_{i=0}^{m-j} PAT[m-i] \cdot n^i \quad ,$$

que é o valor numérico de $PAT[j \cdots m]$.

Por outro lado,

$$A_j = \frac{\sum_{i=0}^{m-1} PAT[m-i] \cdot n^i \quad - \quad \sum_{i=0}^{j-2} PAT[m-i] \cdot n^i}{n^{j-1}} \quad .$$

Logo, se já tivermos calculado o valor de cada $\sum_{i=0}^{k} PAT[m-i] \cdot n^i$, $0 \leq k \leq m-1$, então a operação de comparação entre A_j e B_j passa a ter custo constante. Isto pode ser feito usando a técnica de soma de prefixos [GRY88], da seguinte forma. Primeiro calculamos n^j, $0 \leq j \leq m-1$. Isto pode ser feito em tempo $O(\log m)$ com $O(m/\log m)$ processadores usando a mesma técnica (produto de prefixos). Depois construímos o vetor $H[0 \cdots m-1]$, onde $H[j] = PAT[m-j] \cdot n^j$. A seguir, novamente usando soma de prefixos, construímos $H'[0 \cdots m-1]$, onde $H'[j] = \sum_{i=0}^{j} H[j]$. Note que

$$A_j = \frac{H'[m-1] - H'[j-2]}{n^{j-1}} \quad \text{e} \quad B_j = H'[m-j] \quad .$$

Não basta sabermos se $A_j = B_j$, ou seja, não basta sabermos se $WIT[j] = 0$, mas também o valor exato de $WIT[j]$. No caso em que $A_j = B_j$, temos que $PAT[1 \ldots m-j+1] = PAT[j \ldots m]$ e, portanto, $WIT[j]$ deverá assumir o valor nulo. Mas no caso em que $A_j \neq B_j$, precisamos saber exatamente o valor de $WIT[j]$. Em outras palavras, precisamos saber o valor de qualquer índice w, $1 \leq w \leq m-j+1$, tal que $PAT[w] \neq PAT[j+w-1]$.

Encontraremos um tal w em tempo $O(1)$ da seguinte forma. Seja x o valor de $|A_j - B_j|$ e seja $\bar{x}$ o inteiro x escrito na base n. Sejam $\bar{A}_j$ e $\bar{B}_j$ as representações

de A_j e B_j, respectivamente, na base n. Precisamos descobrir, olhando apenas para x, algum índice t, $1 \leq t \leq m - j + 1$, em $\bar{x}$, tal que $(\bar{A}_j)_t \neq (\bar{B}_j)_t$, ou seja, $PAT[t] \neq PAT[t + j - 1]$. Então $WIT[j] = w$ será igual a t.

Os lemas VI.1 e VI.2 abaixo nos mostram como alcançar tal objetivo.

Lema VI.1 – Seja $\bar{\alpha} = \bar{\alpha}_1 \ldots \bar{\alpha}_r$ a representação do inteiro positivo α na base b e $\bar{\beta} = \bar{\beta}_1 \ldots \bar{\beta}_r$ a representação do inteiro β na base b. Ambos com r dígitos e $\alpha > \beta$. Seja $\lambda = \alpha - \beta$, tal que $\bar{\lambda}$ seja sua representação na base b. Seja t o primeiro índice, da esquerda para a direita, em $\bar{\lambda}$, tal que $\bar{\lambda}_t \neq 0$. Então, se $\bar{\alpha}_t = \bar{\beta}_t$, temos que $\bar{\alpha}_{t-1} \neq \bar{\beta}_{t-1}$.

Prova – Suponhamos que $\bar{\alpha}_t = \bar{\beta}_t$. Então, como $\bar{\lambda}_t \neq 0$, temos que $\bar{\lambda}_t$ foi resultado de

$$(\bar{\alpha}_t - 1)b - \bar{\beta}_t = \bar{\lambda}_t.$$

Logo, o valor de $\bar{\lambda}_{t-1}$ é resultado de

$$\bar{\alpha}_{t-1} - 1 - \bar{\beta}_{t-1} = \bar{\lambda}_{t-1} = 0.$$

Ou seja, $\bar{\alpha}_{t-1} - \bar{\beta}_{t-1} = 1$ e, portanto, $\bar{\alpha}_{t-1} \neq \bar{\beta}_{t-1}$. ∎

Lema VI.2 – Seja $\bar{\alpha}$ a representação do inteiro positivo α na base b, com $\bar{\alpha}$ tendo l dígitos. Então o primeiro dígito não-nulo da esquerda para a direita em $\bar{\alpha}$ é o $(l - \lfloor \log_b \alpha \rfloor)$-ésimo dígito.

Prova – Digamos que $\alpha = \sum_{j=0}^{l-1} \alpha_j b^j$. Seja $t = \max_{\alpha_j \neq 0}\{j\}$. Tal t existe porque $\alpha \neq 0$. Logo,

$$\alpha = \alpha_0 + \alpha_1 b + \alpha_2 b^2 + \cdots + \alpha_t b^t.$$

Como $\alpha_t \neq 0$, temos que $\alpha \geq b^t$. Por outro lado, $\alpha < b^{t+1}$. Daí, $t \leq \log_b \alpha < t+1$. Ou seja, $t = \lfloor \log_b \alpha \rfloor$. Assim, o dígito procurado é o $(l - t)$-ésimo, da esquerda para a direita, em $\bar{\alpha}$. Portanto, vale a afirmação. ∎

Pelo lema VI.2 e pelo fato do tamanho dos padrões comparados ser igual a $m - j + 1$, temos que, no caso em que $x = |A_j - B_j| \neq 0$, o primeiro índice não-nulo, da esquerda para a direita, em $\bar{x}$, é dado por $t = m - j + 1 - \lfloor \log_n x \rfloor$. Então, pelo lema VI.1, temos que $WIT[j] = t$, caso $PAT[t] \neq PAT[t + j - 1]$ ou $WIT[j] = t - 1$, caso contrário. Note que, como no algoritmo de Vishkin, com apenas uma comparação entre dois símbolos descobrimos o valor de $WIT[j]$.

A seguir, a descrição detalhada do algoritmo 1.

início

$F[0] := 1$;

para todo j, $1 \leq j \leq m-1$, **faça, em paralelo** $F[j] := n$;

calcule, em paralelo, o produto dos prefixos de F, de tal forma que

$G[j] = F[0] \cdot \cdots \cdot F[j]$, $0 \leq j \leq m-1$;

para todo j, $0 \leq j \leq m-1$, **faça, em paralelo** $H[j] := PAT[m-j] \cdot G[j]$;

calcule, em paralelo, a soma dos prefixos de H, de tal forma que

$H'[j] := H[0] + \cdots + H[j]$, $0 \leq j \leq m-1$;

$WIT[1] := 0$;

para todo j, $2 \leq j \leq m$, **faça, em paralelo**

início

$x := \left| \frac{H'[m-1] - H'[j-2]}{H[j-1]} - h'[m-j] \right|$;

se $x = 0$

então $WIT[j] := 0$

senão

início

$t := m - j + 1 - \lfloor \log_n x \rfloor$;

se $PAT[t] \neq PAT[t+j-1]$

então $WIT[j] := t$

senão $WIT[j] := t - 1$

fim

fim

fim

Vamos agora mostrar por quê o algoritmo 1 não é ótimo.

Como sabemos [JO90], o custo de uma operação de leitura/escrita de um número N numa célula i da memória de uma PRAM, sendo n o tamanho do problema, é dado por:

$$\lceil 1+\frac{\log N+\log i}{\log n}\rceil .$$

O algoritmo 1, assim como todos os algoritmos descritos neste trabalho, usam no máximo um número polinomial, em n, de células da memória (ou seja, $\log i = O(\log n)$). Logo, o custo de uma operação do algoritmo 1 sobre a memória da PRAM é $O(m)$, já que manipula valores numéricos da ordem de n^m.

Sendo assim, como o algoritmo 1 executa $O(\log m)$ passos, temos que seu tempo paralelo será $O(m \cdot \log m)$ não sendo, portanto, ótimo.

Se nos restringirmos a um alfabeto fixo, fazendo com que os valores numéricos manipulados sejam $O(c^m)$, onde $c = |\Sigma|$, então basta que $m = O(\log n)$, para que se tenha a complexidade ótima desejada.

Em outras palavras, o algoritmo 1 não é ótimo para o caso geral, mas o é para os casos em que o alfabeto é fixo e $m = O(\log n)$.

Vamos agora descrever o algoritmo 2, que tem a complexidade desejada, ou seja, usa tempo paralelo $O(\log m)$, com $O(\frac{n}{\log m})$ processadores de uma CREW-PRAM, para os casos em que o alfabeto é fixo. Porém, necessita de uma condição mais fraca do que aquela imposta ao algoritmo 1, qual seja, a de que $m = O(\log^2 n)$, ou equivalentemente, $\sqrt{m} = O(\log n)$.

Como o problema do algoritmo 1 está relacionado ao fato de estarmos manipulando valores numéricos que representam cadeias muito longas de símbolos, o algoritmo 2 compacta cadeias menores, mais precisamente de tamanho $\sqrt{m}$. Assim, o custo da operação de escrita/leitura do algoritmo 2 na memória da PRAM terá custo $O(1)$, já que os valores numéricos são da ordem de $c^{\sqrt{m}}$ e $O(\log c^{\sqrt{m}}) = O(\sqrt{m} \cdot \log c) = O(\sqrt{m}) = O(\log n)$.

Vamos supor que se deseja calcular $WIT[j]$, $2 \leq j \leq m$ (lembre-se que só precisamos de $WIT[j]$, para $2 \leq j \leq \frac{m}{2}+1$, mas para efeito de complexidade assintótica, podemos calcular para $2 \leq j \leq m$). Devemos então comparar $PAT[j \ldots m]$ com $PAT[1 \ldots m-j+1]$.

Vamos dividir as cadeias $PAT[j \ldots m]$ e $PAT[1 \ldots m-j+1]$ em $s = O(\frac{m-j+1}{\sqrt{m-j+1}}) = O(\sqrt{m-j+1})$ sub-cadeias de $O(\sqrt{m-j+1})$ símbolos cada.

Usando a técnica de soma de prefixos e, em tempo paralelo $O(\log m)$, com $O(\frac{\sqrt{m}}{\log m})$ processadores, calcule o valor numérico de cada bloco de tamanho, digamos, $\lceil \sqrt{m-j+1} \rceil$ de $PAT[j \ldots m]$. No total (para $2 \leq j \leq m$) são usados

$$O\left(\frac{\sqrt{m}}{\log m} \cdot \sqrt{m} \cdot m\right) = O\left(\frac{m^2}{\log m}\right) = O\left(\frac{\log^4 n}{\log m}\right) = O\left(\frac{n}{\log m}\right)$$

processadores, já que $\log^4 n = O(n)$. Faça o mesmo para os blocos de tamanho $\lceil\sqrt{m-j+1}\rceil$ de $PAT[1 \dots m-j+1]$.

Temos agora $PAT[j \dots m]$ representada pelos s números $A_1, A_2, \dots, A_s$ e a cadeia $PAT[1 \dots m-j+1]$ representada pelos s números $B_1, B_2, \dots, B_s$. Note que, com a operação feita acima, já temos os valores numéricos de A_j e B_j, $1 \le j \le s$.

Calcule agora $C_r = A_r - B_r$, $1 \le r \le s$. Todos estes cálculos podem ser feitos em paralelo, num total de $O(\sqrt{m})$ operações para cada j, $2 \le j \le m$. Assim, temos um total de $O(m\sqrt{m}) = O(\log^3 n) = O(n)$ operações. Pelo teorema de Brent, esses cálculos podem ser feitos em tempo $O(\log m)$ com $\frac{n}{\log m}$ processadores.

Para calcularmos $WIT[j]$ devemos saber se $\exists k$, $1 \le k \le s$, tal que $C_k \ne 0$. Se tal k não existir, é porque $A_r = B_r$, $1 \le r \le s$. Em outras palavras, $PAT[j \dots m] = PAT[1 \dots m-j+1]$ e, portanto, $WIT[j]$ deve assumir o valor nulo. Caso exista tal k, é porque $A_k \ne B_k$, ou seja, os blocos correspondentes a A_k e B_k são diferentes. Note que qualquer k nestas condições nos interessa, em particular o mais a esquerda.

Descubra, então, usando o método da árvore binária balanceada [GRY88],

$$k = \min_{1 \le r \le s} \{r \mid A_r \ne B_r\} = \min_{1 \le r \le s} \{r \mid C_r \ne 0\}.$$

Isto pode ser feito em tempo $O(\log m)$ com $O(\frac{\sqrt{m}}{\log m})$ processadores, para cada j, $2 \le j \le m$. No total, o número de processadores necessários é

$$O\left(\frac{\sqrt{m} \cdot m}{\log m}\right) = O\left(\frac{n}{\log m}\right) \quad .$$

Se não existir tal k, então faça $WIT[j] = 0$. Caso contrário, significa que o bloco correspondente a A_k é diferente do bloco correspondente a B_k. Daí o fato de C_k ser diferente de zero.

Encontre, com um processador, um índice t de C_k, tal que $(A_k)_t \ne (B_k)_t$, e faça $WIT[j] = t$. Para esta busca serão necessários $m\sqrt{m} = O(n)$ operações, para $2 \le j \le m$. Assim, pode ser feita em tempo $O(\log m)$ com $O(\frac{n}{\log m})$ processadores.

A diferença básica entre o algoritmo 1 e o algoritmo 2 é que o segundo usa o fato de n ser, em geral, maior que m. Para calcular o WIT, usa-se então n processadores, ao invés de m. Para isso tivemos que ter n em função de m.

Note que mesmo assim n é qualquer. Além disso é preciso que o alfabeto seja de tamanho fixo, para que os valores numéricos manuseados não ultrapassem a $c^{\sqrt{m}}$. Note ainda que, em nenhum momento, precisamos de escrita concorrente.

Considerando o alfabeto de tamanho fixo e $\sqrt{m} \le k \cdot \log n$, o algoritmo de Vishkin para a análise do padrão ainda precisa de tempo paralelo $O(\log^2 m)$ e $O(\frac{m}{\log^2 m})$ processadores para o modelo CREW-PRAM.

Finalmente, a análise do padrão termina com o cálculo do tamanho P do menor período de PAT. Para isso, encontre em $WIT[2 \dots \frac{m}{2}+1]$, o menor índice

k, tal que $WIT[k] = 0$. Isto pode ser feito em tempo paralelo $O(\log m)$ com $O(\frac{m}{\log m})$ processadores de uma CREW-PRAM. Se tal k não existir, então PAT é aperiódico.

Com isso temos um algoritmo para o problema de casamento de cadeias de tempo paralelo $O(\log m)$ e $O(\frac{n}{\log m})$ processadores de uma CREW-PRAM, que funciona para os casos em que o alfabeto é fixo e $m = O(\log^2 n)$. Basta para isso utilizarmos o algoritmo 2, descrito neste capítulo, para a análise do padrão, e a análise de texto de [V85], que tem a mesma complexidade neste mesmo modelo.

Capítulo VII

Conclusões

Abordamos, neste trabalho, o problema de casamento de cadeias sob o ponto de vista paralelo.

Os algoritmos seqüenciais considerados mais importantes pela literatura, apresentados de forma sucinta no capítulo II, refletem a dificuldade de paralelização de algoritmos seqüenciais para o problema. Tanto é que os algoritmos paralelos mais conhecidos não se baseiam nos seqüenciais.

Entre os algoritmos paralelos, podemos destacar, além do apresentado em [G85], que foi o primeiro algoritmo paralelo a usar as propriedades de periodicidade, o algoritmo apresentado em [V85] que, com a interessante idéia dos duelos, direcionou as técnicas usadas por quase todos os algoritmos surgidos depois.

O algoritmo de [KLP], que é uma aplicação de um algoritmo desenvolvido para resolver o problema de sufixo-prefixo, tem uma interessante peculiaridade. O píoprio algoritmo para sufixo-prefixo pode ser usado para calcular o vetor WIT. Basta o executarmos tendo $A = B = PAT$ como entrada. Assim, temos um algoritmo ótimo para casamento de cadeias, de tempo paralelo $O(\log m)$ e $O(\frac{n}{\log m})$ processadores, muito mais simples que o de Vishkin [V85].

Os algoritmos apresentados em [V90] e [G92] são interessantes, sob o ponto de vista técnico, no sentido de que usam idéias importantes (a exemplo de [V90], que introduz a idéia da assinatura). Mas, por outro lado, não podem incluir em seus custos o que foi gasto no pré-processamento. Assim, dos pontos de vista algorítmico e de complexidade, eles têm pouco valor.

Podemos então dizer que o algoritmo de Breslauer e Galil [BG90], que leva tempo paralelo $O(\log \log m)$ é o que melhor se pode obter, em termos de complexidade de tempo paralelo.

Quando se fala em algoritmos para problemas que possuem muitas aplicações práticas, como é o caso dos algoritmos para casamento de cadeias, logo

se tem em mente procurar desenvolver algoritmos que, na prática, tenham bom desempenho.

Por outro lado, quando se fala em algoritmos para modelos de PRAM, não se leva em conta esses aspectos práticos, já que estes modelos são de difícil implementação .

Nosso algoritmo, proposto no capítulo anterior, se encontra no centro desta questão, já que funciona bem para casos mais freqüentes na prática (alfabeto de tamanho fixo e $m = O(\log^2 n)$).

De qualquer maneira, para estes casos particulares, ele tem um desempenho bastante razoável (melhor, por exemplo, que a de [V85]), já que funciona em tempo $O(\log m)$, com $O(n/\log m)$ processadores de uma CREW-PRAM. Como sabemos, o tempo paralelo esperado para qualquer algoritmo para casamento de cadeias para o modelo CREW-PRAM, hoje, é $O(\log m \cdot \log\log m)$. Isto se dá em função do fato de se ter uma perda logarítmica no melhor tempo paralelo conhecido para se resolver o problema de casamento de cadeias, que aliás é ótimo, devido a [BG90] e [BG91].

Referências Bibliográficas

[**ACO**] Aho,A.V. and Corasick,M.J. *Efficient String Matching: An Aid to Bibliographic Search.* Communicarions of The ACM 18,333-340 (1975).

[**AHU**] Aho,A.V., Hopcroft,J.E. and Ullman,J.D. *The Design and Analysis of Computer Algorithms.* Addison Wesley (1974).

[**AKL**] Akl,S. *The Design and Analysis of Parallel Algorithms.* Prentice Hall (1989).

[**BG90**] Breslauer,D. and Galil,Z. *An Optimal* $O(\log \log n)$ *Parallel String Matching Algorithm.* SIAM J.Comput. 19,1051-1058 (1990).

[**BG91**] Breslauer,D. and Galil,Z. *A Lower Bound for Parallel String Matching.* Proc. 23nd ACM Symp. on Theory of Computation, 439-443 (1991).

[**BM**] Boyer,R.S. and Moore,J.S. *A Fast String Searching Algorithm.* Communications of The ACM 20,762-772 (1977).

[**CGG**] Colussi,L., Galil,Z. and Giancarlo,R. *On The Exact Complexity of String Matching.* 31th IEEE FOCS, 135-143 (1990).

[**EPG**] Epstein,D. and Galil,Z. *Parallel Algorithmic Techniques For Combinatorial Computation.* Ann. Rev. Comput. Sci., 233-283 (1988).

[**GS0**] Galil,Z.& Seiferas,J.*Saving Space in Fast String-Matching.* SIAM J.Comp. 9,417-438 (1980).

[**GS**] Galil,Z. and Seiferas,J. *Time-Space-Optimal String Matching.* J. of Computer and System Sciences 26,280-294 (1983).

[**G85**] Galil,Z. *Optimal Parallel Algorithms for String Matching.* Information and Control 67, 144-157 (1985).

[**G92**] Galil,Z. *A Constant-Time Optimal Parallel String-Matching Algorithm* Comunicação privada (1992).

[**JO90**] Johnson,D.S. *A Catalog of Complexity Classes* Handbook of Theoretical Computer Science (Jan Van Leeuwen), Volume A. Elsevier, 67-161 (1990).

[**KAR**] Karp,R.M. and Ramachandran V. *Parallel Algorithms for Shared-Memory Machines.* Tech. Rep. Univ. of California-Berkley (1989).

[**KLP**] Kedem,Z., Landau,G. and Palem,K. *Optimal Parallel Suffix-Prefix Matching Algorithm and Applications* Comunicação privada (1991).

[**KMP**] Knuth,D.E., Morris,J.H. & Pratt,V.B. *Fast Pattern Matching in Strings.* SIAM J. Comput. 6,323-350 (1977).

[**MCDP**] Crochemore,M. and Perrim,D. *Two-Way String-Matching.* J. ACM 38, 651-675 (1991).

[**SHI81**] Shiloach,Y. and Vishkin,U. *Finding the Maximum, Merging, and Sorting in a Parallel Computation Model* Journal of Algorithms 2,88-102 (1981).

[**V85**] Vishkin,U. *Optimal Parallel Pattern Matching in Strings.* Information and Control 67, 91-113 (1985).

[**V90**] Vishkin,U. *Deterministic Sampling - A New Technique for Fast Pattern Matching.* Proc. 22nd ACM Symp. On Theory of Computation, 170-179 (1990).