Algoritmos CGM para Busca Uni e Bidimensional de Padrões com e sem Escala

Henrique Mongelli

TESE APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE DOUTOR
EM
CIÊNCIA DA COMPUTAÇÃO

Orientador: **Prof. Dr. Siang Wun Song**

Durante a elaboração deste trabalho o autor recebeu apoio financeiro da CAPES

— São Paulo, abril de 2000 —

Algoritmos CGM para Busca Uni e Bidimensional de Padrões com e sem Escala

Este exemplar corresponde à redação
final da tese devidamente corrigida
e defendida por Henrique Mongelli
e aprovada pela comissão julgadora.

São Paulo, 04 de abril de 2000.

Banca Examinadora:

- Prof. Dr. Siang Wun Song (orientador) (IME-USP)
- Prof. Dr. José Augusto Ramos Soares (IME-USP)
- Profa. Dra. Maria Cristina Silva Boeres (UFF)
- Prof. Dr. Edson Norberto Cáceres (UFMS)
- Prof. Dr. Edil Severiano Tavares Fernandes (UFRJ)

à minha esposa Magda
e à minha filha Letícia

Agradecimentos

Ao meu orientador, Prof. Dr. Siang Wun Song, pela dedicação e paciência.

À minha mãe Alzira, ao meu pai Orlando, à minha sogra Mercedes e ao meu sogro Bento (*in memoriam*), aos meus irmãos, cunhados e sobrinhos, pelo carinho e apoio.

Aos professores Dilma Menezes da Silva, José Coelho de Pina Júnior, José Augusto Ramos Soares, Marco Dimas Gubitoso, Markus Endler, Yoshiharu Kohayakawa.

Aos funcionários da Seção de Informática/IME, ao Sr. Vladimir do Serviço de Materiais, Compras e Patrimônio e às secretárias da Diretoria/IME.

Ao Marco Aurélio Stephanes pela ajuda e pelas trocas de idéias.

Aos colegas do Departamento de Ciências Exatas/CEUC/UFMS e Departamento de Computação e Estatística/CCET/UFMS.

A todos que contribuíram, direta ou indiretamente, no desenvolvimento deste trabalho.

Resumo

Dados um texto e um padrão, o problema de busca de padrões em textos consiste em determinar as posições do texto onde existe uma ocorrência do padrão. Quando texto e padrão são cadeias de caracteres, a busca é dita unidimensional. Quando ambos são matrizes, a busca é dita ser bidimensional. Existem variações deste problema onde se permite a busca do padrão, de alguma maneira, modificado. A modificação que permitiremos ao nosso padrão é que ele possa estar escalado. Descrevemos algoritmos seqüencias lineares para estes problemas, uni ou bidimensionais, com e sem escala, presentes na literatura. Para o caso bidimensional sem escala é apresentado, ainda, um algoritmo de tempo sublinear sob determinadas condições nas matrizes de entrada.

Para estes problemas propomos novos algoritmos paralelos, utilizando o modelo CGM (*Coarse Grained Multicomputers*), cujos tempos de computação local são lineares na entrada (local), consomem memória também linear e utilizam apenas uma rodada de comunicação em que são trocados, no máximo, uma quantidade também linear de dados. As condições do modelo são, assim, respeitadas. Do nosso conhecimento, não há na literatura outros algoritmos paralelos em modelos de granularidade grossa para o problema de busca unidimensional com escala e para os problemas de busca bidimensional com ou sem escala.

Estes algoritmos propostos foram implementados em linguagem C, utilizando *interface* PVM e foram executados na máquina *Parsytec PowerXplorer*. Os resultados experimentais obtidos mostraram que as implementações tiveram ganhos significativos ao utilizar-se mais de um processador.

Abstract

Given a text and a pattern, the problem of pattern matching consists of determining all the positions of the text where the pattern occurs. When the text and the pattern are strings, the matching problem is said to be unidimensional. When both are matrices, the matching is termed bidimensional. There are variations of this problem where we allow the matching using a somehow modified pattern. A modification that we will allow is that the pattern can be scaled. We describe sequential linear algorithms known in the literature, for both uni and bidimensional problems, with and without scale. For the bidimensional case, without scale, we describe also an algorithm using sublinear time under certain conditions on the matrix input.

For these problems, we propose new parallel algorithms, under the CGM (Coarse Grained Multicomputer) model, with local computing time linear in the input, requiring linear memory and use only one communication round, during which at most a linear amount of data are exchanged. The conditions of the model are thus satisfied. For the unidimensional scaled matching problem and the bidimensional matching with or without scale, there are no coarse-grained parallel algorithms, to our knowledge, known in the literature.

These proposed algorithms were implemented in C, using the PVM interface and were executed on the Parsytec PowerXplorer. The experimental results obtained showed a significant speedup using more than one processor.

Sumário

Notação/Nomenclatura

Na tabela a seguir enumeramos algumas notações e nomenclaturas utilizadas neste trabalho, referenciando as páginas onde cada uma delas aparece pela primeira vez. Algumas descrições mais extensas ou complexas não foram incluídas podendo ser obtidas na página de referência. Alguns itens possuem mais de uma página de referência por terem seu significado de alguma forma alterado, neste caso, cada página indica a primeira aparição do item com o novo significado.

Not./Nomencl.	Descrição	Ref.
$w \sqsubset x$	a cadeia w é um prefixo da cadeia x	pág. 6
$w \sqsupset x$	a cadeia w é um sufixo da cadeia x	pág. 6
Árvore Cartesiana	ver texto	pág. 6
Árvore de Sufixos	ver texto	pág. 7
LCA	*Lowest Common Ancestor*	pág. 11
Mínimo Intervalar	ver texto	pág. 13
π	vetor obtido com o pré-processamento do padrão	pág. 24
σ	vetor obtido com o pré-processamento do padrão	pág. 32
Fronteira	ver texto	pág. 32 pág. 62
S^{Σ}	parte de símbolos da representação fatorada de uma cadeia S	pág. 44
$S^{\#}$	parte de expoentes da representação fatorada de uma cadeia S	pág. 44
$\hat{n}$	número de símbolos da representação fatorada	pág. 44
$\bar{P}$	cadeia onde cada símbolo é uma linha da matriz P	pág. 58
C	cadeia com os dados de entrada	pág. 57 pág. 79 pág. 102
ST	árvore sufixa da cadeia C	pág. 57 pág. 79 pág. 102

Not./Nomencl.	Descrição	Ref.
bloco	ver texto	pág. 76
coluna potência	ver texto	pág. 77
FT	representação fatorada da matriz texto T	pág. 77
FP	representação fatorada da matriz padrão P	pág. 77
$B_l[i,c]$	comprimento do prefixo comum mais longo das linhas i e $i+1$ começando na coluna c	pág. 79
$B_r[i,c]$	comprimento do prefixo comum mais longo das linhas i e $i+1$ terminando na coluna $c-1$	pág. 79
$CT_l[c]$	árvore Cartesiana para o vetor $B_r[c]$	pág. 80
$CT_r[c]$	árvore Cartesiana para o vetor $B_l[c]$	pág. 80
R	cadeia onde cada símbolo é uma linha da matriz padrão P	pág. 80
FR	representação fatorada da cadeia R	pág. 80
$\hat{F}R$	cadeia FR descartando-se o primeiro e o último elementos	pág. 80
$\bar{\pi}$	ver texto	pág. 63 pág. 89
$\bar{\sigma}$	ver texto	pág. 64 pág. 89
$\bar{\pi}'$	ver texto	pág. 64 pág. 89
$\bar{\sigma}'$	ver texto	pág. 64 pág. 89
Q	lista de finais de blocos	pág. 81
FR_j	ver texto	pág. 88
$\hat{F}R_j$	ver texto	pág. 88
FR'_j	ver texto	pág. 88
$\hat{F}R'_j$	ver texto	pág. 88
$\bar{Q}$	ver texto	pág. 89
Q_s	ver texto	pág. 89
$\bar{Q}_s$	ver texto	pág. 89
Q_n	ver texto	pág. 89
$\bar{Q}_n$	ver texto	pág. 89
k-bloco	ver texto	pág. 101
árvore k-altura Cartesiana	ver texto	pág. 102
$D_l[i,c]$	número de repetições do símbolo em $T[i,c]$, iniciando em $[i,c]$ e para a esquerda	pág. 103
$D_r[i,c]$	número de repetições sucessivas do símbolo em $T[i,c]$, iniciando na posição $[i,c]$ e para a direita	pág. 103

Not./Nomencl.	Descrição	Ref.
grupo de linhas	ver texto	pág. 104
âncora	ver texto	pág. 105
I_j^k	listas de intervalos na coluna j que podem conter k-ocorrências	pág. 105
Q_j^k	listas de finais de k-blocos	pág. 107

Capítulo 1

Introdução e Preliminares

1.1 Introdução

Neste trabalho, apresentamos algoritmos paralelos eficientes, no modelo CGM (*Coarse Grained Multicomputers*), para os problemas de buscas unidimensional e bidimensional de padrões, com e sem escala.

O problema de busca unidimensional de padrões consiste em: dados um vetor texto T de comprimento n e um vetor P de comprimento m, $m \leq n$, cujos elementos pertencem a um alfabeto finito Σ, determinar as posições do texto em que os subvetores de comprimento m, começando nestas posições, sejam iguais a P. O algoritmo mais conhecido para este problema foi apresentado por Knuth, Morris e Pratt [41], é denominado algoritmo KMP e resolve o problema em tempo $O(n+m)$. Outros algoritmos podem ser encontrados em [17, 30, 38, 42, 50]. Aplicações para este problema são encontradas em processamento de textos, reconhecimento de fala, visão computacional e biologia molecular [37].

Este problema pode ser estendido ao caso bidimensional, onde temos uma matriz texto T de dimensão $n_l \times n_c$ e uma matriz padrão P, de dimensão $m_l \times m_c$, com $m_l \leq n_l$ e $m_c \leq n_c$, com elementos pertencentes a um alfabeto finito Σ. Para estas matrizes estamos interessados em determinar todas as posições de T, onde as submatrizes de dimensão $m_l \times m_c$, começando nestas posições, sejam iguais a P. Algoritmos seqüenciais para este problema, lineares no tamanho da entrada, podem ser encontrados em [3, 7, 9, 10, 11, 14, 20, 21, 23, 29, 34, 35]. Aplicações de buscas bidimensionais são encontradas em bibliotecas multimídia, visão computacional e busca de dados heterogêneos [6].

Por uma questão de simplicidade, consideramos matrizes texto e padrão quadradas de ordem n e m, respectivamente, sendo que os algoritmos funcionam também para os casos de matrizes retangulares. O tamanho de uma matriz é o seu número de elementos. Assim, as matrizes texto terão tamanho n^2 e as matrizes padrão terão tamanho m^2.

Para alguns problemas de busca de padrões em imagens, queremos buscar não só o padrão, mas também algumas variações deste. Vamos considerar o caso em que o padrão pode estar

presente no texto com alterações em suas dimensões, mais especificamente, consideraremos que cada dimensão do padrão pode estar multiplicada por um inteiro k, que denominaremos escala. Em uma imagem, as escalas de um padrão podem ocorrer, por exemplo, por causa da distância da fonte de captação ou pelo uso de *zooms*.

Neste caso, o problema é determinar todas as ocorrências da matriz padrão P na matriz texto T, para todas as escalas k. Uma idéia inicial é fazer buscas de padrões para cada escala; porém, isto leva a um tempo superlinear. Amir *et al* [8] apresentaram um algoritmo principal que determina todas estas ocorrências para cada escala k em tempo linear no tamanho da entrada. Apresentaram, ainda, um algoritmo de tempo linear para a busca unidimensional com escala e dois algoritmos para a busca bidimensional sem escala: um de tempo linear e outro de tempo sublinear, sob algumas condições nos dados de entrada.

Na Figura 1.1 ilustramos um exemplo de matrizes texto e padrão com ocorrências do padrão no texto com escalas 1 e 2.

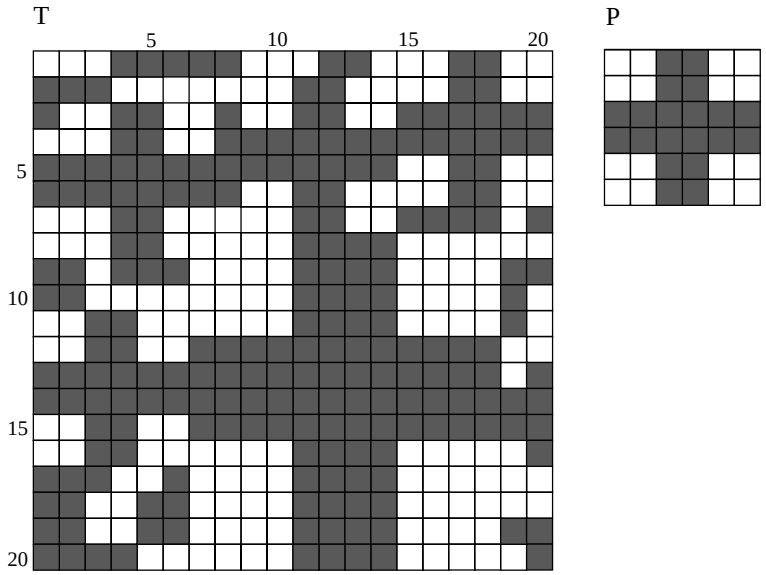

Figura 1.1: Exemplo de figuras armazenadas nas matrizes texto T e padrão P com ocorrências de P em T nas posições $[1, 15]$, $[2, 9]$, $[3, 2]$, $[11, 1]$ com escala 1 e na posição $[8, 7]$ com escala 2.

A partir da descrição de Amir *et al* [8] desenvolvemos um algoritmo paralelo para este problema no modelo CGM (*Coarse Grained Multicomputers*) de memória distribuída.

Este modelo é composto por um conjunto de p processadores com suas memórias locais que se comunicam através de uma rede de interconexão que permite comunicação ponto-a-ponto entre os processadores.

Um algoritmo no modelo CGM, ou simplesmente algoritmo CGM, para um problema que tenha entrada de tamanho n e que utilize p processadores é descrito através de superpassos compostos por uma rodada de computação local e uma rodada de comunicação. Na rodada de computação são manipulados os dados armazenados localmente. Na rodada de comunicação são

trocadas informações entre os processadores. O modelo impõe algumas restrições: a memória utilizada não pode exceder $O(n/p)$ e em cada rodada de comunicação cada processador pode enviar e/ou receber no máximo $O(n/p)$ dados. Neste modelo, nosso objetivo é minimizar o número de rodadas de comunicação e a quantidade de computação local.

O algoritmo CGM, proposto neste trabalho, para o caso bidimensional com escala tem tempo de computação local linear e utiliza apenas uma rodada de comunicação, respeitando a quantidade máxima de dados na comunicação imposta pelo modelo. Além deste algoritmo, obtivemos também algoritmos CGM para as buscas unidimensional sem e com escala e para a busca bidimensional sem escala. Estes algoritmos também rodam em tempo local linear e utilizam apenas uma rodada de comunicação.

Estes algoritmos foram implementados em linguagem C utilizando *interface* PVM e executados na máquina paralela *PowerXPlorer* de arquitetura grade bidimensional com 16 processadores. Os resultados obtidos se mostraram bastante relevantes, levando a *speedups* bons, mesmo a conexão entre processadores sendo tipo *Ethernet*.

Apesar de no modelo CGM não estarmos preocupados diretamente com a quantidade de dados trocados em cada rodada de comunicação (desde que respeitando o limite imposto pelo modelo), mas com a quantidade de rodadas de comunicação, os algoritmos descritos trocam uma quantidade de informações que depende da quantidade de possíveis posições de ocorrência, como pode ser visto nos algoritmos a serem discutidos.

Nos algoritmos CGM desenvolvidos minimizamos o número de rodadas de comunicação (uma rodada), a quantidade de dados comunicados (igual ao número de possíveis ocorrências) e a quantidade de computação local (linear no tamanho da entrada local). As implementações destes algoritmos mostraram resultados experimentais promissores, obtendo *speedups* próximos ao ótimo, para padrões pequenos.

Para os problemas de busca de padrões sem escala, existem algoritmos paralelos, no modelo PRAM, que estão descritos em [22, 33, 40, 42, 53] para o caso unidimensional e descritos em [4, 5, 19, 22] para o caso bidimensional.

Em modelos de granularidade grossa, não é de nosso conhecimento a existência de outros resultados para o problema de busca unidimensional com escala e de buscas bidimensionais com e sem escala. Publicações recentes relacionadas com este trabalho foram apresentadas por: Mongelli e Song [46, 47], para o problema de mínimo intervalar (*range minima*), que utilizamos no algoritmo CGM para busca bidimensional com escala (Capítulo 4); Ferragina e Luccio [31], para o problema de buscas múltiplas em cadeias no modelo BSP e para outros modelos de granularidade grossa; Boxer *et al* [15], para o problema de reconhecimento de padrões geométricos, no modelo CGM. Ainda relacionado a este trabalho, Amir *et al* [6] apresentaram um algoritmo seqüencial para o problema de busca unidimensional de padrões com escala real.

Os algoritmos CGM apresentados, neste trabalho, para os problemas de busca de padrões são inéditos, não sendo encontrados, na literatura, algoritmos em modelos de memória distribuída para estes problemas. Contribuímos, ainda, com o algoritmo CGM para o problema de mínimo intervalar [46, 47], que é utilizado no algoritmo CGM para o problema de busca bidimensional com escala.

Neste primeiro capítulo, apresentamos também uma descrição mais completa do modelo CGM, os conceitos de Árvore Cartesiana, Árvore de Sufixos, do problema de LCA (*Lowest Common Ancestor*) e do problema de determinação do comprimento do prefixo mais longo de dois sufixos de uma cadeia dada, apresentando uma noção de como resolvê-los seqüencialmente. Descrevemos ainda o problema de Mínimo Intervalar, utilizado no algoritmo de busca bidimensional de padrões com escala, e os algoritmos seqüencial e CGM para resolvê-lo. Também enumeramos as operações sobre listas duplamente ligadas que serão utilizadas nos algoritmos e descrevemos o ambiente computacional em que os algoritmos foram implementados e executados, gerando os dados experimentais.

No Capítulo 2, descrevemos o algoritmo KMP e sua versão CGM, o algoritmo de busca unidimensional com escala e o correspondente algoritmo paralelo CGM. Para os algoritmos CGM mostramos os resultados experimentais obtidos.

No Capítulo 3, apresentamos dois algoritmos para o caso bidimensional sem escala e suas correspondentes versões CGM . O primeiro algoritmo é linear no tamanho da entrada e o segundo é sublinear sob algumas condições. Ambos os algoritmos no modelo CGM rodam em tempo linear e utilizam uma única rodada de comunicação. Os resultados experimentais destes algoritmos, obtidos a partir da implementação de cada um deles, são mostrados nesse capítulo.

O Capítulo 4 contém a descrição do algoritmo de Amir *et al* [8] para o caso bidimensional com escala. Descrevemos o correspondente algoritmo CGM e, ao final do capítulo, apresentamos os resultados obtidos com sua implementação.

No Capítulo 5, apresentamos conclusões e sugestões de trabalhos futuros.

1.2 Modelo CGM

Em 1990, Valiant [52] introduziu o modelo BSP (*Bulk Synchronous Parallel*) e o conceito de que a computação paralela deveria ser modelada como uma série de superpassos em vez de passos individuais de troca de mensagens ou acessos à memória compartilhada. Todo algoritmo BSP consiste de uma seqüência de tais superpassos, cada um consistindo de várias operações de troca de mensagens (ou acessos à memória compartilhada). Os superpassos são separados por barreiras de sincronização. Toda mensagem enviada em um superpasso está disponível para o recebedor somente no passo subseqüente. O modelo BSP tornou a computação paralela de granularidade grossa e levou a uma redução substancial no *overhead* de sincronização. Neste modelo, uma h-relação em um superpasso corresponde ao envio e/ou recebimento de no máximo h mensagens em cada processador.

O modelo CGM (*Coarse Grained Multicomputers*), proposto por Dehne *et al* [27] adicionou a restrição de comunicação de granularidade grossa. Um algoritmo CGM é um caso especial de um algoritmo BSP onde todas as operações de comunicação de um superpasso são feitas em uma h-relação com $h \leq n/p$.

Assim, um CGM(n, p) consiste de p processadores, onde cada processador tem $O(n/p)$ de memória local. Os processadores podem estar conectados por qualquer meio de interconexão.

A memória local é tipicamente maior que $O(1)$ ($n/p \geq p$). Esta característica dá ao modelo o nome de *coarse grained* (granularidade grossa). Na Figura 1.2 temos uma representação do modelo CGM. Um algoritmo CGM consiste de rodadas (*rounds*) alternadas de computação local e comunicação global. Em uma rodada de comunicação uma h-relação (com $h = O(n/p)$) é roteada, isto é, cada processador envia $O(n/p)$ dados e recebe $O(n/p)$ dados. O tempo para um algoritmo CGM é a soma dos tempos para as rodadas de computação e comunicação.

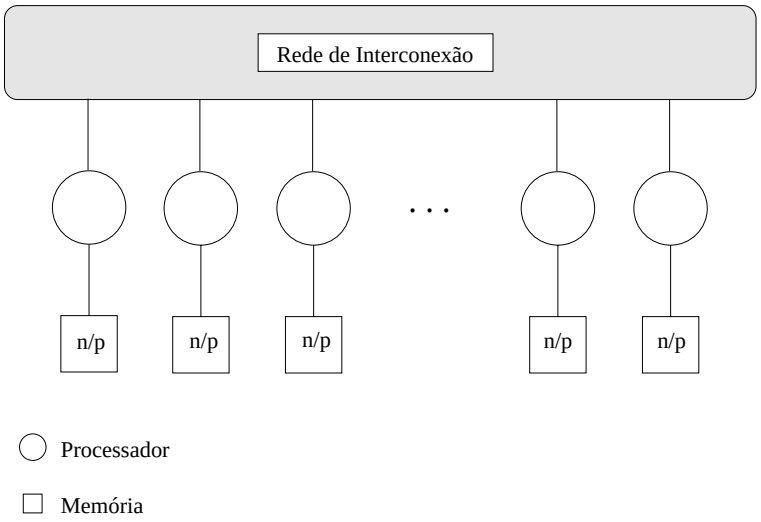

Figura 1.2: Representação do modelo CGM para p processadores, considerando entradas de tamanho n.

Os algoritmos CGM, quando implementados em multiprocessadores atualmente disponíveis, se comportam bem e exibem *speedups* similares àqueles previstos em suas análises [24]. Para estes algoritmos, o maior objetivo é minimizar o número de superpassos e a quantidade de computação local.

1.3 Listas

Na descrição dos algoritmos utilizaremos listas duplamente ligadas. Sobre estas listas será possível efetuar as seguintes operações:

- a criação de uma lista vazia feita através da atribuição do conjunto vazio à lista;
- a inserção de um elemento i no início de uma lista L se dará pela operação $Insere(i, L)$;
- a operação $Remove(L)$ remove o primeiro elemento da lista L;
- $Elemento(L)$ obtém o primeiro elemento da lista L; e
- $Próximo(L)$ aponta para o próximo elemento da lista L.

Cada nó de uma lista é formado pelos campos que contêm os apontadores para os elementos anterior e posterior da lista. Os demais campos, contendo as informações, vão depender dos algoritmos onde as listas serão utilizadas.

1.4 Cadeias de Caracteres

Vamos supor que um texto está representado em um vetor indexado de 1 a n e um padrão é um vetor indexado de 1 a m, $n \geq m$, cujos elementos são caracteres (ou símbolos) sobre um alfabeto Σ dado. Assim, consideremos T um vetor texto e P um vetor padrão com $T[i] \in \Sigma$ e $P[j] \in \Sigma$, $1 \leq i \leq n$, $1 \leq j \leq m$. Estes vetores serão também denominados *cadeias de caracteres*, podendo ser representadas por seqüências de caracteres; por exemplo, $T = \tau_1\tau_2 \ldots \tau_n$ e $P = \rho_1\rho_2 \ldots \rho_m$. Utilizaremos, indistintamente, qualquer uma destas duas representações. Assim teremos $T[1] = \tau_1, \ldots, T[n] = \tau_n$ e $P[1] = \rho_1, \ldots, P[m] = \rho_m$.

Definição 1 *A cadeia $aaa\cdots a$, denotada por a^k, onde o símbolo a é repetido k vezes, é denominada* ***escala da cadeia de um único*** **a** ***por um fator multiplicativo*** **k**, *ou simplesmente como* **a** ***escalado a*** **k**. *Analogamente, consideremos a cadeia $A = a_1 \cdots a_l$.* **A** ***escalado a*** **k**, *denotado A^k, é a cadeia $a_1^k \cdots a_l^k$.*

Definição 2 *Seja $P = (p_{i,j})$, onde $1 \leq i, j \leq m$, uma matriz bidimensional sobre um alfabeto finito Σ.* **P** ***escalada a*** **k** *é a matriz $km \times km$ que resulta da substituição de cada símbolo $P_{i,j}$ de P por uma matriz $k \times k$ cujos elementos são todos iguais ao símbolo $p_{i,j}$. Isto é,*

$$P^k[i,j] = P\left[\left\lceil \frac{i}{k} \right\rceil, \left\lceil \frac{j}{k} \right\rceil\right].$$

Denominamos Σ^* o conjunto de todas as cadeias de comprimento finito formadas utilizando-se o alfabeto Σ. A cadeia vazia será denotada por ε, e temos que $\varepsilon \in \Sigma^*$. O comprimento de uma cadeia x é denotado por $|x|$. Dadas duas cadeias x e y, a concatenação destas cadeias, contendo $|x| + |y|$ caracteres, é representado por xy e consiste dos caracteres de x seguidos pelos caracteres de y.

Dizemos que uma cadeia w é um *prefixo* de uma cadeia x, denotado $w \sqsubset x$, se $x = wy$ para alguma cadeia $y \in \Sigma^*$. Notemos que se $w \sqsubset x$ então $|w| \leq |x|$. Analogamente, dizemos que uma cadeia w é um *sufixo* de uma cadeia x, denotado $w \sqsupset x$, se $x = yw$ para alguma cadeia $y \in \Sigma^*$. A cadeia vazia ε é ao mesmo tempo prefixo e sufixo de qualquer cadeia. Notemos, ainda, que para quaisquer duas cadeias x e y e qualquer caracter a, temos $x \sqsupset y$ se e somente se $xa \sqsupset ya$ e que, também, $\sqsubset$ e $\sqsupset$ são relações transitivas.

1.5 Árvore Cartesiana

Esta estrutura de dados foi apresentada por Vuillemin [54]. A definição de árvore Cartesiana é dada a seguir. A construção desta estrutura a partir de um vetor é feita em tempo seqüencial

linear. Esta estrutura é útil no algoritmo seqüencial do Mínimo Intervalar (seção 1.8) e nos algoritmos seqüenciais de Busca Bidimensional sublinear sem escala (seção 3.4.1) e com escala (seção 4.2).

Definição 3 (Árvore Cartesiana [54]) *Dado um vetor $A_{0,n-1} = (a_0, a_1, \ldots, a_{n-1})$ de n números reais distintos, a árvore Cartesiana de $A_{0,n-1}$ é uma árvore binária com nós rotulados com os elementos do vetor A. A raiz da árvore tem rótulo $a_m = \min\{a_0, a_1, \ldots, a_{n-1}\}$. Sua subárvore esquerda é a árvore Cartesiana de $A_{0,m-1} = (a_0, a_1, \ldots, a_{m-1})$, e sua árvore direita é a árvore Cartesiana de $A_{m+1,n-1} = (a_{m+1}, \ldots, a_{n-1})$. A árvore Cartesiana de um vetor vazio é a árvore vazia.*

Na Figura 1.3 apresentamos um exemplo de uma árvore Cartesiana para um vetor particular.

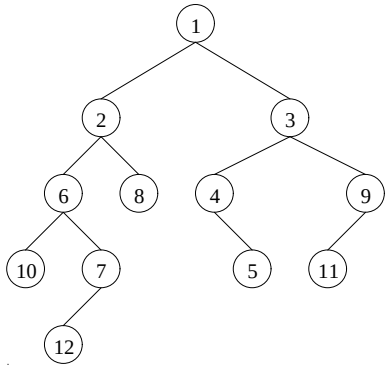

Figura 1.3: Árvore Cartesiana correspondente ao vetor $(10, 6, 12, 7, 2, 8, 1, 4, 5, 3, 11, 9)$.

A árvore Cartesiana para um vetor $A_{0,n-1} = (a_0, a_1, \ldots, a_{n-1})$ é obtida construindo-se uma seqüência de árvores T_i, $i = 0, \ldots, n-1$, onde T_i é a árvore Cartesiana para o vetor $A_{0,i}$. T_i é obtida de T_{i-1}, subindo-se nesta árvore pelo caminho mais à direita até encontrarmos o nó cujo rótulo a_k seja menor que a_i. A subárvore direita de a_k torna-se a subárvore esquerda de a_i e a_i torna-se filho direito de a_k. O tempo total para construir a árvore é $O(n)$, uma vez que um nó deixa de pertencer ao caminho mais à direita assim que ele é visitado.

1.6 Árvores de Sufixos

Em muitos problemas que envolvam manipulação de cadeias podemos utilizar estruturas de dados obtidas do pré-processamento de uma cadeia dada S, de forma que consultas sobre subcadeias de S possam ser processadas rapidamente. A descrição a seguir é baseada em JáJá [37].

Uma estrutura de dados possível é uma *árvore digital de busca* (ou *trie*) T associada a um conjunto de cadeias distintas $C_1, C_2, \ldots, C_m$ sobre um alfabeto Σ. Vamos supor que cada C_i não é um prefixo de algum C_j com $i \neq j$. A árvore digital de busca T é uma árvore com raiz com m nós terminais tal que:

1. Cada aresta de T está rotulada com um símbolo do alfabeto Σ, e está orientada no sentido raiz para nós terminais.
2. Quaisquer duas arestas que saem de um mesmo nó têm rótulos diferentes.
3. Cada folha u corresponde a uma e somente uma cadeia C_i, de forma que a concatenação dos rótulos das arestas do caminho da raiz até u seja C_i.

Nosso interesse é construir tal árvore considerando os sufixos de uma cadeia S dada. Para garantirmos que cada sufixo estará representado unicamente por uma folha em T, o último caracter da cadeia S não pode aparecer em qualquer outra posição de S. Dessa forma, nenhum sufixo de S é um prefixo de um sufixo diferente de S. Assim, quaisquer dois sufixos de S tomarão, eventualmente, caminhos distintos em T até atingir sua folha correspondente. Para que toda cadeia S possua este último caracter distinto dos demais caracteres de S, vamos considerar que toda cadeia S possua um caracter final que não aparece antes.

Uma árvore digital de busca T construída a partir de S pode ser muito grande. Por exemplo, se considerarmos a cadeia $S = a^n b^n\$$, ela terá $\Omega(n^2)$ nós.

Utilizaremos uma versão compacta de uma árvore digital de busca, chamada *árvore de sufixos* de forma a utilizar espaço linear. Uma árvore de sufixos T representando uma cadeia S obedece às seguintes restrições:

1. Uma aresta de T pode representar qualquer subcadeia não-vazia de S.
2. Cada nó não terminal de T, exceto a raiz, deve ter no mínimo dois arcos filhos.
3. As cadeias representadas por nós irmãos de T devem começar com caracteres diferentes.

Na Figura 1.4 temos um exemplo de uma árvore de sufixos construída para a cadeia *ababbaaba*\$.

A construção de uma árvore de sufixos pode ser feita por algoritmos que levam tempo linear, como descrito por Weiner [55], e tempo e espaço lineares no tamanho da cadeia de entrada, como descrito por McCreight [45]. A seguir vamos descrever o segundo algoritmo, desenvolvido por McCreight [45].

Vamos representar cada símbolo de um alfabeto Σ por letras romanas minúsculas e seqüências finitas de caracteres (eventualmente vazias) por letras gregas minúsculas.

Um *caminho parcial* é uma seqüência de arestas vizinhas da árvore que começa na raiz em direção aos nós terminais.

Um *caminho* é um caminho parcial que termina em uma folha.

Garantimos que um caminho parcial representará unicamente uma subcadeia de S obtida pela concatenação das cadeias em suas arestas.

O *locus* de uma cadeia é um nó no fim de um caminho parcial que representa esta cadeia.

Uma *extensão* de uma cadeia α é qualquer cadeia cujo prefixo seja α.

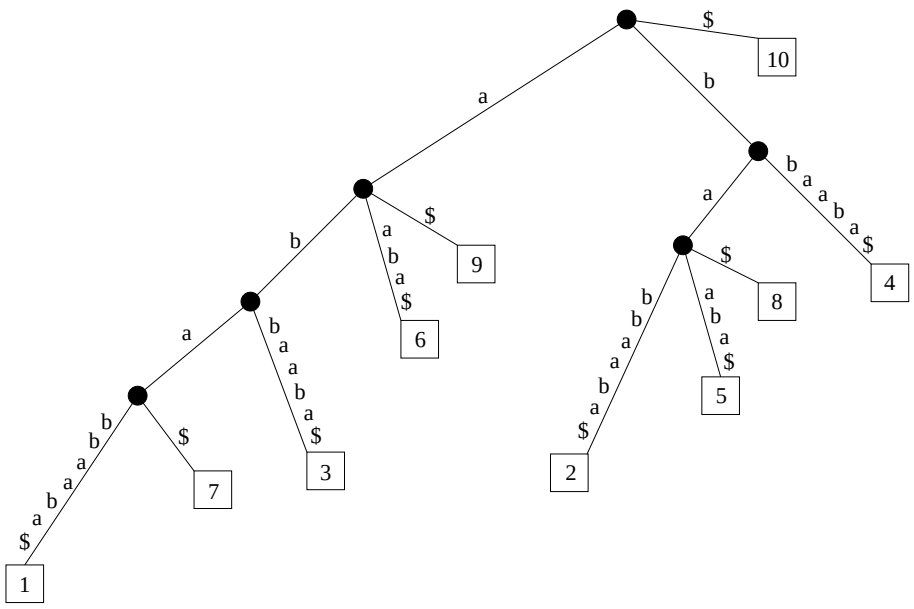

Figura 1.4: Árvore de sufixos correspondente à cadeia *ababbaaba*\$.

O *locus estendido* de uma cadeia α é o locus da menor extensão de α para o qual o locus esteja definido.

O *locus contraído* de uma cadeia α é o locus do maior prefixo de α para o qual o locus esteja definido.

O algoritmo inicia com uma árvore vazia T_0 e insere caminhos correspondentes a sufixos de S um de cada vez, do sufixo mais longo para o mais curto.

Em uma iteração i construiremos a árvore T_i a partir da árvore T_{i-1}. Definimos suf_i como sendo o sufixo de S, começando na posição i ($suf_1 = S$). Neste passo inserimos um caminho correspondente a suf_i na árvore T_{i-1} obtendo T_i. Definimos $head_i$ como sendo o mais longo prefixo de suf_i que é prefixo de suf_j para algum $j < i$. Equivalentemente, $head_i$ é o mais longo prefixo de suf_i cujo locus estendido exista na árvore T_{i-1}. Definimos $tail_i$ como sendo $suf_i - head_i$. Para inserirmos suf_i em T_{i-1}, o locus estendido de $head_i$ em T_{i-1} é encontrado, um novo nó não-terminal é construído para dividir a aresta que entra e torna-se o locus de $head_i$ se necessário, e finalmente uma nova aresta rotulada $tail_i$ é construída daquele nó não terminal para uma nova folha.

Para que o algoritmo seja eficiente, temos que utilizar uma estrutura de dados para representar a árvore T que também seja eficiente. A árvore será montada de tal forma que cada nó tenha apontadores para o (primeiro) filho e para seus irmãos. Além disso, cada aresta representa uma subcadeia de S que será armazenada no nó de chegada desta aresta por um par de índices denotando as posições inicial e final em S.

Com esta representação, após a determinação de $head_i$ em T_{i-1}, a introdução de um novo nó não terminal e de uma nova aresta correspondente a $tail_i$ leva tempo constante. Como são

inseridos n sufixos, a determinação de $head_i$ em T_{i-1} também deve levar tempo constante (em média sobre todos os passos).

Para isto, utilizaremos a seguinte relação entre $head_{i-1}$ e $head_i$.

Lema 1 *Se $head_{i-1}$ pode ser escrito como $x\delta$ para algum caracter s e alguma (possivelmente vazia) cadeia δ, então δ é prefixo de $head_i$.*

A demonstração deste lema está descrita em [45]. Para explorarmos esta propriedade, incluímos ligações auxiliares. De cada nó não terminal que é o locus de $x\delta$, uma ligação *sufixo* é introduzida apontando para o locus de δ. Esta ligação permite ao algoritmo no passo i pegar um atalho para o locus de $head_i$ a partir do locus de $head_{i-1}$ visitado no passo anterior.

O algoritmo é descrito em três passos:

Passo A Primeiramente, identificam-se três cadeias χ, α e β, com as seguintes propriedades:

1. $head_{i-1}$ pode ser representado por $\chi\alpha\beta$.
2. se o locus contraído de $head_{i-1}$ em T_{i-2} não é a raiz, ele é o locus de $\chi\alpha$. Caso contrário, α é vazio.
3. χ é uma cadeia de no máximo um caracter que é vazia somente se $head_{i-1}$ é vazia.

O lema 1 garante que $head_i$ pode ser representado como $\alpha\beta\gamma$ para alguma (possivelmente vazia) cadeia γ. Temos dois casos:

α vazio: denomina-se c a raiz e vai-se para o passo B.

α não vazio: por definição, o locus de $\chi\alpha$ deve ter existido na árvore T_{i-2}. Segue-se sua ligação sufixo para um nó não terminal c e vai-se para o passo B.

Notemos que em ambos os casos c é o locus de α.

Passo B Este passo é chamado *rescanning*, e é o passo chave do algoritmo. Como $\alpha\beta\gamma$ é definido como $head_i$, pela definição de *head* sabemos que o locus estendido de $\alpha\beta$ existe na árvore T_{i-1}. Isto significa que existe um caminho a partir de c (locus de α) que é uma extensão de β. Isto significa que existe uma seqüência de arestas a partir do nó c que representa alguma extensão de β. Para percorrer β encontra-se a aresta filha de c que representa uma subcadeia ρ cujo primeiro símbolo seja igual ao primeiro símbolo de β, esta aresta leva a um nó que chamaremos f. Então, comparam-se os comprimentos de β e ρ. Se o comprimento de β for maior, faz-se uma chamada recursiva de $\beta - \rho$ iniciando no nó f. Se β tem comprimento menor ou igual, então β é um prefixo de ρ, e foi encontrado o locus estendido de $\alpha\beta$. Esta fase está terminada e leva tempo linear no número de nós encontrados. Um novo nó não terminal é construído para ser locus de $\alpha\beta$, se tal nó ainda não existir. O locus de $\alpha\beta$ é referenciado como d e vai-se para o passo C. (Notar que um novo nó para ser o locus de $\alpha\beta$ é construído somente se γ é vazio.)

Passo C Este passo é chamado de *scanning*. Inicialmente, verifica-se se a ligação sufixo do locus de $\chi\alpha\beta$ não está definida. Em caso afirmativo, esta ligação aponta para d. Inicia-se uma busca do locus estendido de $\alpha\beta\gamma$ a partir de d. A maior diferença entre esta fase e a anterior é que na anterior conhecia-se o comprimento de β de antemão, enquanto aqui não se conhece, previamente, o comprimento de γ. Assim, deve-se percorrer a árvore de acordo com os caracteres de $tail_{i-1}$, da qual γ é um prefixo, um a um da esquerda para a direita. Quando "cai-se" fora da árvore, o locus estendido de $\alpha\beta\gamma$ foi encontrado. O último nó de T_{i-1} encontrado neste percurso é o locus contraído de $head_i$ em T_{i-1}. Um novo nó não terminal é construído para ser o locus de $\alpha\beta\gamma$ se tal não existir. E, finalmente, uma nova aresta com rótulo $tail_i$ é construída do locus de $\alpha\beta\gamma$ para uma nova folha, finalizando a iteração i.

O tempo de execução do algoritmo depende também do tamanho do alfabeto da cadeia de entrada. Para um alfabeto de tamanho fixo, o algoritmo descrito obtém uma árvore de sufixos em tempo e espaço lineares no tamanho da cadeia de entrada. Uma descrição de como lidar com outros alfabetos pode ser encontrada em [45].

1.7 Problema do Ancestral Comum Mais Baixo (LCA)

Definição 4 (Ancestral Comum Mais Baixo) *O **ancestral comum mais baixo** de dois vértices u e v de uma árvore com raiz é o vértice w que é um ancestral de u e v e que está mais distante da raiz. $w = LCA(u, v)$.*

Não estamos interessados somente no problema de dado um par de vértices u e v determinar $w = \mathrm{LCA}(u, v)$. O que queremos é um caso mais geral e aplicável em uma série de problemas como descrito em Reif [49]. Estamos interessados em, dada uma árvore $T = (V, E)$ com raiz, pré-processar T de forma que uma consulta $\mathrm{LCA}(u, v)$, para um par qualquer de vértices u e v de T, possa ser respondida rapidamente - em tempo seqüencial $O(1)$.

Para este problema, conhecido como Problema do Ancestral Mais Baixo (LCA – *Lowest Common Ancestor*), temos dois casos especiais:

1. se T é simplesmente um caminho, basta determinarmos a distância de cada vértice à raiz, o que nos permite responder a $\mathrm{LCA}(u, v)$ em tempo constante, comparando-se as distâncias de u e v à raiz.
2. T é uma árvore binária completa.

1.7.1 O Algoritmo para uma Árvore Binária Completa

Neste caso, dada uma árvore binária completa T com raiz, inicialmente determina-se o número *in-ordem* dos vértices de T. O número *in-ordem* dos vértices de uma árvore corresponde à ordem dos vértices obtida em um percurso *in-ordem* na árvore a partir da raiz. Isto pode ser feito em

tempo linear no número de nós da árvore. A partir disto, cada vértice será identificado através do seu número *in-ordem* e convenientemente dado em sua representação binária. Indexamos os bits na representação binária do bit menos significativo, que tem índice 0, para o mais significativo, que tem índice l. Dados dois vértices x e y da árvore T, seja i a posição do bit mais à esquerda em que x e y diferem. LCA(x, y) consiste dos $l - i$ bits mais à esquerda, seguidos por um "1" e i "0"s. Assim, após a determinação dos números *in-ordem*, podemos responder a consultas LCA(x, y) em tempo $O(1)$.

1.7.2 O Algoritmo para uma Árvore Arbitrária

O algoritmo descrito a seguir foi desenvolvido por Schieber e Vishkin [51] e também pode ser encontrado em Reif [49]. A descrição deste algoritmo foi feita para um modelo PRAM EREW, rodando em tempo $O(\log n)$, usando $n/\log n$ processadores. A versão seqüencial deste algoritmo, que utilizamos nas implementações, roda em tempo e espaço lineares no número de nós da árvore e é mais eficiente que o algoritmo descrito em [36].

Neste caso, reduziremos, através do pré-processamento, nosso problema aos casos especiais já vistos. Na etapa de pré-processamento, a árvore T inicial é decomposta em cadeias vértice-disjuntas. Dessa forma, para responder a uma consulta LCA(x, y) temos dois casos possíveis:

1. x e y pertencem a uma mesma cadeia. Neste caso, basta usar os níveis dos vértices pré-calculados.
2. Sejam C_x e C_y as cadeias da decomposição de T que contêm x e y, respectivamente. Na etapa de pré-processamento determinamos um mapeamento injetivo do conjunto de cadeias no conjunto de vértices de uma árvore binária completa B com $2^{\lfloor \log\ n \rfloor + 1} - 1$ vértices.

Este mapeamento tem a seguinte propriedade:

Propriedade 1 (Propriedade de Preservação de Ascendência) *Sejam v e u dois vértices quaisquer em T, e sejam C_v e C_u as cadeias que contêm v e u, respectivamente. Se v é um ancestral de u, então C_v é mapeado em um ancestral do vértice no qual C_u é mapeado em B.*

Dada uma consulta LCA(x, y), usamos o algoritmo para árvores binárias completas para calcular o LCA (em B) dos dois vértices em que C_x e C_y foram mapeados. Assim, com mais alguma informação adicional, podemos identificar a cadeia em T que contém LCA(x, y) e, finalmente, podemos obtê-lo.

A etapa de pré-processamento consiste em determinar para cada vértice v da árvore T os rótulos: $LEVEL(v)$, $INLABEL(v)$ e $ASCENDANT(v)$; e uma tabela de consulta chamada $HEAD$. Estes rótulos e a tabela de consulta podem ser determinados em tempo linear no número de nós da árvore. O rótulo $LEVEL(v)$ corresponde ao nível do nó v na árvore. O rótulo $INLABEL(v)$ contém o valor de um descendente de v cujo *bit* 1 mais à direita tenha o maior índice dentre os descendentes de v. O número $INLABEL$ decompõe a árvore em cadeias

vértice-disjuntas compostas por todos os nós com o mesmo valor de $INLABEL$. O número $ASCENDANT(v)$ armazena todos os ancestrais de $INLABEL(v)$ na árvore binária completa B que também são números $INLABEL$ de ancestrais de v em T. A tabela $HEAD$ contém o vértice mais alto em cada cadeia $INLABEL$, isto é, $HEAD(k)$ armazena o vértice mais próximo da raiz dentre aqueles nós com número $INLABEL$ igual a k. De posse destes dados, podemos efetuar uma consulta LCA em tempo seqüencial constante. Os detalhes podem ser obtidos em Schieber e Vishkin [51] ou Reif [49].

1.8 Problema de Mínimo Intervalar

Definição 5 *Dado um vetor $A = (a_0, a_1, \ldots, a_{n-1})$ de números reais, definimos $MIN(i,j) = \min\{a_i, \ldots a_j\}$. O **problema de mínimo intervalar (range minima)** consiste em pré-processar o vetor A de forma que consultas $MIN(i,j)$ possam ser respondidas em tempo constante, para todo $0 \leq i, j \leq n-1$.*

Consultas de mínimo intervalar serão utilizadas no Capítulo 4. Estas consultas poderão ser nos dados locais ou em dados contidos em processadores diferentes. Para o caso seqüencial, usaremos o algoritmo de Gabow *et al* [32]. No caso dos dados em processadores diferentes, utilizaremos o algoritmo CGM de Mongelli e Song [47, 46]. Estes algoritmos encontram-se descritos a seguir.

O algoritmo para o problema de mínimo intervalar no modelo CGM, descrito em [47] usa os algoritmos seqüenciais de Gabow *et al* [32] e Alon e Schieber [2] que são executados usando os dados locais em cada processador.

1.8.1 Algoritmo Seqüencial de Gabow *et al*

Este algoritmo usa a estrutura de dados árvore Cartesiana descrita na seção 1.5 e nela é aplicado um algoritmo para o problema do LCA.

O algoritmo é dado a seguir.

ALGORITMO: Mínimo_Intervalar(Gabow et al)
Entrada: o vetor $A = (a_0, a_1, \ldots, a_{n-1})$ de n números reais.
Saída: uma estrutura de dados que responde a consultas $MIN(i,j)$ em tempo constante.

1. Construir a árvore Cartesiana de A.

2. Usar um algoritmo seqüencial linear para o problema LCA, usando a árvore Cartesiana.

fim algoritmo

A construção da árvore Cartesiana leva tempo linear, bem como o algoritmo para o problema do LCA. O algoritmo leva, então, tempo linear para obter a estrutura que responde a consultas de Mínimo Intervalar em tempo constante. Qualquer consulta $MIN(i,j)$ pode ser respondida

da seguinte forma: da definição recursiva de árvore Cartesiana temos que o valor de $MIN(i, j)$ é o valor do LCA de a_i e a_j. Assim, cada consulta de mínimo intervalar pode ser respondida em tempo constante através de uma consulta de LCA em uma árvore Cartesiana. Logo, o problema de mínimo intervalar é solucionado em tempo constante.

1.8.2 Algoritmo Seqüencial de Alon e Schieber

O algoritmo de Alon e Schieber tem complexidade de tempo $O(n \log n)$. Apesar desta complexidade não-linear, este algoritmo é crucial na descrição do algoritmo CGM, como será visto na seção 1.8.3. Sem perda de generalidade, vamos considerar n como sendo uma potência de 2.

Na descrição deste algoritmo, utilizamos o conceito de vetores mínimo prefixo e mínimo sufixo, definidos a seguir, que são casos particulares de vetores de soma prefixa e soma sufixa, respectivamente, considerando o operador mínimo. Uma exposição mais detalhada de somas prefixas e sufixas pode ser obtida em Reif [49].

Definição 6 *Considere um vetor $A = (a_1, a_2, \ldots, a_n)$ de n números reais. O vetor mínimo prefixo é o vetor $P = (\min\{a_0\}, \min\{a_0, a_1\}, \ldots, \min\{a_0, a_1, \ldots, a_{n-1}\})$. O vetor mínimo sufixo é o vetor $S = (\min\{a_0, a_1, \ldots, a_{n-1}\}, \min\{a_1, \ldots, a_{n-1}\}, \ldots, \min\{a_{n-2}, a_{n-1}\}, \min\{a_{n-1}\})$.*

ALGORITMO: Mínimo_Intervalar(Alon e Schieber)
Entrada: o vetor $A = (a_0, a_1, \ldots, a_{n-1})$ de n números reais.
Saída: uma estrutura de dados que responde a consultas $MIN(i, j)$ em tempo constante.

1. Construir uma árvore binária completa T com n folhas.
2. Associar os elementos de A às folhas de T.
3. Para cada nó v de T, calcular P_v e S_v, os vetores de mínimo prefixo e mínimo sufixo, respectivamente, das folhas da subárvore com raiz v.

fim algoritmo

A árvore T construída pelo algoritmo *Mínimo_Intervalar(Alon e Schieber)* será denominada árvore-PS. A Figura 1.5 ilustra uma árvore-PS gerada por este algoritmo.

Vamos introduzir outra notação.

Notação 1 *Considere uma árvore binária completa e um nó v da árvore. Denotemos $a_r, a_{r+1}, \ldots, a_s$ as folhas abaixo de v que são descendentes de v. Para cada j tal que $r \leq j \leq s$, definimos*

$$Pos(a_j) = j - r.$$

O processamento das consultas $MIN(i, j)$ com $0 \leq i, j \leq n-1$ é feito como se segue. Determinamos $w = \text{LCA}(a_i, a_j)$ em T. Sejam v e u os filhos esquerdo e direito de w, respectivamente. Então o valor de $MIN(i, j)$ é o mínimo entre o valor de $S_v[Pos(a_i)]$ e o valor de $P_u[Pos(a_j)]$.

A árvore-PS é uma árvore binária completa e, como vimos na seção 1.7, consultas LCA em árvores binárias completas podem ser feitas em tempo constante.

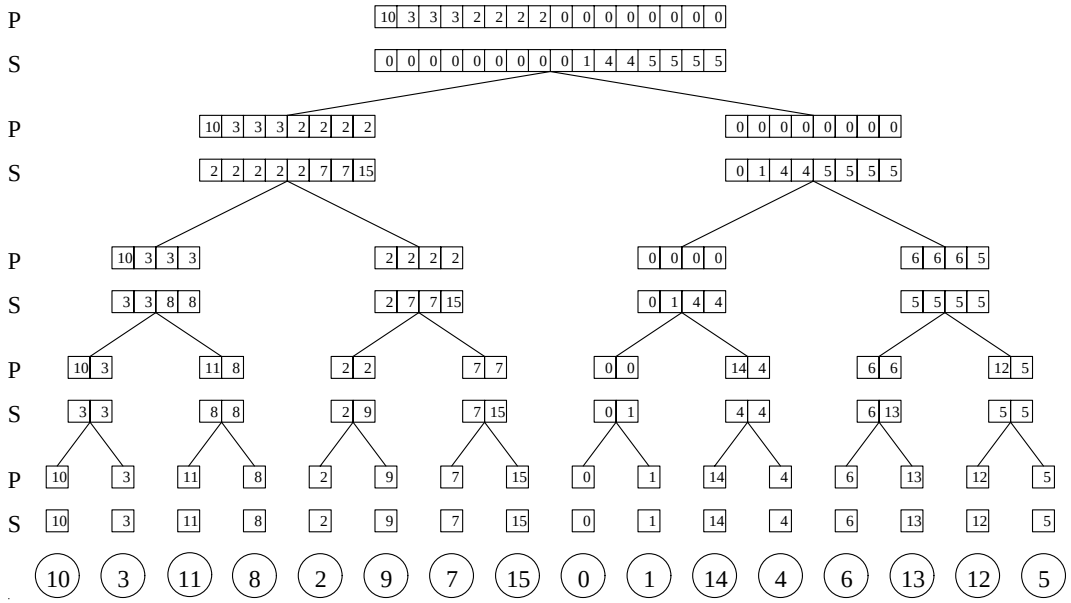

Figura 1.5: árvore-PS gerada pelo Algoritmo *Minimo_Intervalar(Alon e Schieber)* para um vetor particular.

1.8.3 O Algoritmo CGM

Em um algoritmo CGM, estamos interessados em reduzir o número de rodadas de comunicação e concentrar na computação de dados locais nos processadores.

Baseado nos algoritmos seqüenciais vistos nas seções anteriores, podemos construir um algoritmo CGM para o problema de mínimo intervalar. O algoritmo CGM de Mongelli e Song [46, 47] é executado em tempo $O(n/p)$ e usa $O(1)$ rodadas de comunicação. A maior dificuldade é no armazenamento dos dados necessários pelos processadores de forma que as consultas possam ser feitas em tempo constante sem violar o limite de $O(n/p)$ de memória fixada pelo modelo CGM.

O modelo $CGM(n, p)$ precisa de um mínimo de $O(n/p)$ de memória em cada processador. Infelizmente, não há o que fazer se esta quantidade de memória disponível for menor que este mínimo. Primeiro, precisamos de memória suficiente para armazenar os dados de entrada. Também precisamos de memória para construir as estruturas de dados em cada processador de forma a obter tempo constante nas consultas do mínimo intervalar.

Na descrição do algoritmo, consideraremos a seguinte notação.

Notação 2 *Dado um vetor* $A = (a_0, a_1, \ldots, a_{n-1})$, *escrevemos* $A[i] = a_i$, $0 \leq i, \leq n-1$, *e* $A[i \ldots j] = (a_i, a_{i+1}, \ldots, a_{j-1}, a_j)$, $0 \leq i \leq j \leq n-1$.

A idéia do algoritmo é baseada em como as consultas $MIN(i, j)$ podem ser respondidas. Consideraremos p processadores denotados por $0, 1, \ldots, p-1$. Sem perda de generalidade, assumimos p como sendo um potência de 2. Cada processador armazena n/p posições contíguas

do vetor de entrada. Assim, dado $A = (a_0, a_1, \ldots a_{n-1})$, o processador i armazena o subvetor $A_i = (a_{in/p}, \ldots, a_{(i+1)(n/p)-1})$, para $0 \leq i \leq p-1$. $MIN(i,j)$ é respondido dependendo da localização de a_i e a_j nos processadores. Temos os seguintes casos:

1. Se a_i e a_j estão em um mesmo processador, o domínio do problema se reduz ao subvetor armazenado localmente naquele processador. Assim, precisamos de uma estrutura de dados para responder a este tipo de consulta em tempo constante. Esta estrutura de dados é fornecida em cada processador pelo Algoritmo *Mínimo_Intervalar(Gabow et al)* (página 13).

2. Se a_i e a_j estão em processadores distintos $\bar{i}$ e $\bar{j}$ (sem perda de generalidade, $\bar{i} < \bar{j}$), respectivamente, temos dois subcasos:

   (a) Se $\bar{i} = \bar{j} - 1$, isto é, a_i e a_j estão em processadores vizinhos, $MIN(i,j)$ então corresponde ao mínimo entre o mínimo de a_i até o final do subvetor $A_{\bar{i}}$ e o mínimo do começo do subvetor $A_{\bar{j}}$ até a_j. Estes mínimos podem ser novamente determinados pela estrutura de dados obtida pelo Algoritmo *Mínimo_Intervalar(Gabow et al)* (página 13). Para determinar o mínimo destes mínimos, precisamos de uma rodada de comunicação.

   (b) Se $\bar{i} < \bar{j} - 1$, $MIN(i,j)$ corresponde ao mínimo do subvetor $A_{\bar{i}}[i \ldots (\bar{i}+1)(n/p)-1]$, o mínimo do subvetor $A_{\bar{j}}[\bar{j}(n/p) \ldots j]$ e os mínimos dos subvetores $A_{\bar{i}+1}, \ldots, A_{\bar{j}-1}$. Os primeiros dois mínimos são determinados como no subcaso anterior. Os mínimos dos subvetores $A_{\bar{i}+1}, \ldots A_{\bar{j}-1}$ são facilmente determinados usando a árvore Cartesiana do vetor de mínimos de todos os subvetores A_k, $0 \leq k \leq p-1$. Isto se reduz a um problema de mínimo intervalar restrito a um vetor de p valores. Assim, precisamos de uma estrutura de dados para responder a estas consultas em tempo constante. Como o vetor de mínimos contém apenas p valores, esta estrutura de dados pode ser obtida pelo Algoritmo *Mínimo_Intervalar(Alon e Schieber)* (página 14).

A dificuldade do Caso 2(b) é que não podemos construir a árvore-PS explicitamente em cada processador como descrito na seção 1.8.2, pois para isto seria necessário uma memória de tamanho $O(p \log p)$, maior do que o fixado pelo modelo CGM, que é $O(n/p)$, com $n/p \geq p$. Para contornar esta dificuldade, armazenamos apenas informações parciais na árvore-PS em cada processador. Construímos vetores P' e S' de $\log p + 1$ posições em cada processador, como descrito a seguir.

Vamos descrever esta construção para um processador i, com $0 \leq i \leq p-1$. Seja b_i o valor do mínimo do subvetor A_i. Seja v um nó qualquer da árvore T tal que a subárvore com raiz v tem b_i como folha, e seja d_v a *profundidade* de v em T, isto é, o comprimento do caminho da raiz até v, como definido em [1]; e seja l_v o *nível* de v, que é a altura da árvore menos a profundidade de v, como definido em [1] ($l_v = \log p - d_v$, pois a árvore tem altura $\log p$). O vetor P' (respectivamente, S') contém na posição l_v o valor do vetor P_u (respectivamente, S_v), do nível l_v de T, na posição correspondente à folha b_i. Em outras palavras, temos $P'[l_v] = P_v[i \bmod 2^{l_v}]$.

A Figura 1.6 ilustra a correspondência entre os vetores P' e S' armazenados em cada processador e a árvore-PS construída pelo algoritmo *Mínimo_Intervalar(Alon e Schieber)*. Na Figura 1.6(b), as posições em destaque nos vetores S em cada nível da árvore correspondem ao mínimo sufixo de $b_0 = 3$ em cada nível. Desta forma, obtemos o vetor S' no processador 0 (Figura 1.6(c)).

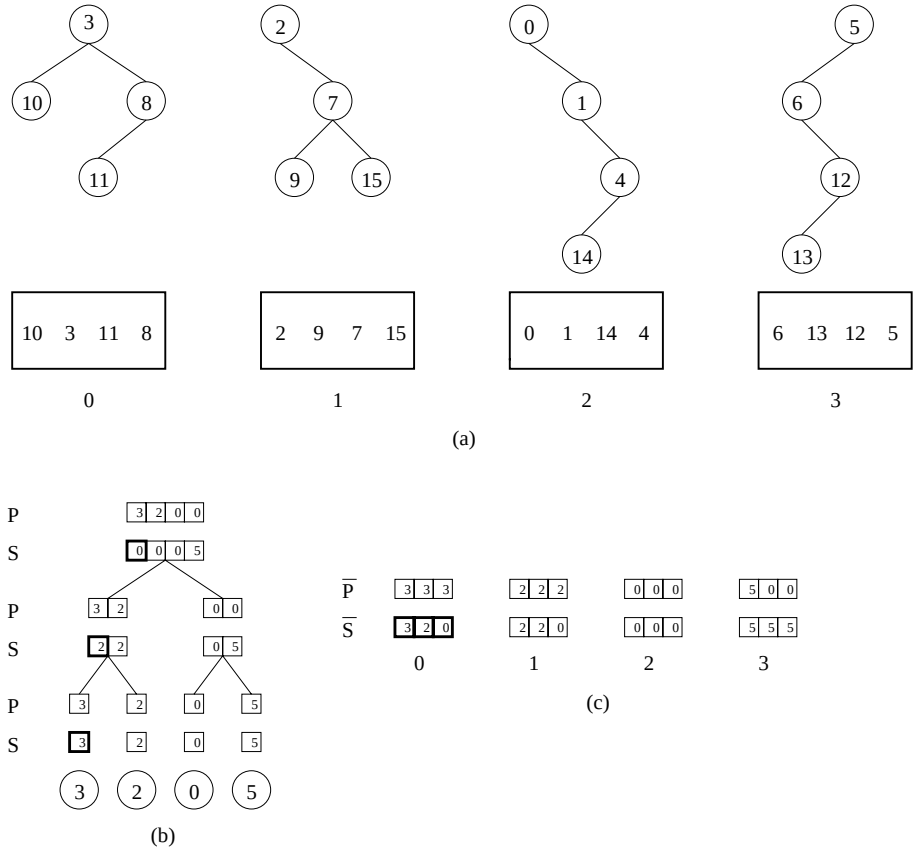

Figura 1.6: Execução do Algoritmo *Mínimo_Intervalar(CGM)* usando o vetor (10, 3, 11, 8, 2, 9, 7, 15, 0, 1, 14, 4, 6, 13, 12, 5). (a) Os dados distribuídos nos processadores e as árvores Cartesianas correspondentes. (b) Árvores-PS construídas pelo Algoritmo *Mínimo_Intervalar(Alon e Schieber)* para o vetor $(3, 2, 0, 5)$ de mínimos dos processadores. (c) Vetores P' e S' construídos pelo passo 3 do Algoritmo *Mínimo_Intervalar(CGM)* correspondente aos vetores P e S de T.

A seguir, damos uma descrição em alto nível do algoritmo.

ALGORITMO: Mínimo Intervalar(CGM)
Entrada: o vetor $A = (a_0, a_1, \ldots, a_{n-1})$ com n números reais.
Saída: uma estrutura de dados que responde a consultas $MIN(i, j)$ em tempo constante.
{Cada processador recebe n/p posições contíguas do vetor A, dividido em subvetores $A_0, A_1, \ldots, A_{p-1}$.}

1. Cada processador i executa seqüencialmente o Algoritmo *Mínimo_Intervalar(Gabow et al)*.

2. Comentário: Cada processador constrói um vetor $B = (b_i)$ de tamanho p, contendo os mínimos dos subvetores armazenados em cada processador.

   2.1. Cada processador i calcula $b_i = \min A_i = \min\{a_{i(n/p)}, \ldots, a_{(i+1)(n/p)-1}\}$.

   2.2. Cada processador i envia b_i aos outros processadores.

   2.3. Cada processador i coloca em b_k o valor recebido do processador k, $k \in \{0, \ldots, p-1\} \setminus \{i\}$.

3. Cada processador i executa os procedimentos *Constrói_P'* and *Constrói_S'* (ver a seguir) .

fim algoritmo

Dado um vetor B, o seguinte procedimento constrói o vetor P' no processador i, para $0 \leq i \leq p-1$. Este procedimento constrói P' em tempo $O(p)$ $(= O(n/p))$ usando somente dados locais. A construção do vetor S' é feita de maneira simétrica, considerando B na ordem inversa. Na descrição dos algoritmos, a indentação dos comandos delimita o corpo do laço. Esta convenção é utilizada na descrição de todos os demais algoritmos e procedimentos.

PROCEDIMENTO: Constrói_P'.
Entrada: o veto $B = (b_0, b_1, \ldots, b_{p-1})$ com p números reais.
Saída: o vetor P' of $\log p + 1$ posições.
1. $P'[0] \leftarrow b_i$
2. $aponta \leftarrow i$
3. $inordem \leftarrow 2 * i + 1$
4. **para** $k \leftarrow 1$ **até** $\log p$ **faça**
5. $\quad P'[k] \leftarrow P'[k-1]$
6. $\quad$ **se** $\lfloor inordem/2^k \rfloor$ é ímpar
7. $\quad\quad$ **então para** $l \leftarrow 1$ **até** 2^{k-1} **faça**
8. $\quad\quad\quad aponta \leftarrow aponta - 1$
9. $\quad\quad\quad$ **se** $P'[k] > B[aponta]$
10. $\quad\quad\quad\quad$ **então** $P'[k] \leftarrow B[aponta]$

fim procedimento

Para simplificar a prova de corretude deste procedimento, consideramos p como sendo uma potência de 2 e que, em cada processador i, o vetor B contém as folhas de uma árvore binária completa, como na descrição do Algoritmo *Mínimo_Intervalar(Alon e Schieber)*. Não armazenamos inteiramente o vetor em cada nó interno desta árvore, mas somente as $\log p + 1$ posições do vetor de mínimo prefixo em cada nível correspondendo à posição i do vetor B.

O valor da variável *inordem* da linha 3 é o valor do número *in-ordem* da folha contendo b_i, obtido em uma travessia *in-ordem* nos nós da árvore. Pode ser visto facilmente que estes valores, da esquerda para a direita, são números ímpares no intervalo $[1, 2p-1]$.

Teorema 1 *O Procedimento Constrói_P' corretamente calcula o vetor P', para cada processador i, $0 \leq i \leq p-1$.*

Prova. Para um processador i, $0 \leq i \leq p-1$, provamos o seguinte invariante em cada iteração do laço da linha 4:

Em cada iteração k, seja T_k a subárvore que contém b_i e tem raiz no nível k. $P'[k]$ armazena a posição i do vetor de mínimo prefixo do subvetor de B correspondente às folhas da subárvore T_k e a variável *aponta* contém o índice, em B, da folha de T_k mais à esquerda.

Inicialmente, provamos que o invariante vale antes da primeira iteração. Neste caso, $k = 0$, $P'[k] = b_i$ e $aponta = i$ são válidos, pois a subárvore T_k é formada apenas pela folha que contém b_i e está na posição i de B.

Vamos supor que o invariante seja válido imediatamente antes do início de uma iteração k e vamos mostrar que permanece válido no fim desta iteração.

Na iteração k, seja C_k o subvetor de B cujos elementos são as folhas da subárvore T_k com raiz v. T_k contém b_i e v está no nível k.

Analisamos dois casos:

1. $\lfloor inordem/2^k \rfloor$ é par.

   Como estamos considerando árvores binárias completas, o valor par de $\lfloor inordem/2^k \rfloor$ significa que a folha correspondente a b_i está na subárvore esquerda de v.

   Neste caso, os elementos à esquerda de b_i em C_k são os mesmos elementos à esquerda de b_i em C_{k-1}. Assim, $P'[k] = P'[k-1]$. Além disso, a folha mais à esquerda de T_k é também a folha mais à esquerda da árvore T_{k-1} que contém b_i. Assim, o valor da variável *aponta* não muda.

   No Procedimento *Constrói_P'*, a linha 5 faz a atualização $P'[k] \leftarrow P'[k-1]$, e este valor de $P'[k]$ não é alterado até o final da iteração. O valor da variável *aponta* também não muda.

   Assim, neste caso, o invariante permanece válido no final da iteração k.

2. $\lfloor inordem/2^k \rfloor$ é ímpar.

   Neste caso, a folha correspondente a b_i está na subárvore direita de raiz v.

   O valor armazenado em $P'[k-1]$ corresponde ao mínimo prefixo do subvetor C_{k-1} do vetor B. A variável *aponta* armazena o índice do vetor B, correspondendo à folha mais à esquerda da subárvore cujas folhas são os elementos de C_{k-1}. Isto é, esta variável aponta para a primeira posição de C_{k-1}.

   O subvetor C_k, considerado na iteração k, corresponde às folhas da subárvore T_k. Assim, este subvetor é formado pelas folhas das subárvore esquerda e direita de v. (As folhas da subárvore direita correspondem aos elementos do subvetor C_{k-1}.)

   Como este novo subvetor C_k apresenta valores à esquerda de b_i que não estavam envolvidos na determinação de $P'[k-1]$, o valor de $P'[k]$ será o mínimo entre $P'[k-1]$ e os valores correspondentes às folhas da subárvore esquerda de v.

O laço **para** da linha 7 examina estas posições, comparando os valores com $P'[k]$ (inicialmente igual a $P'[k-1]$) e atualizando-o quando necessário. Isto é feito usando a variável *aponta*, que varre estas posições, e, ao final, aponta para a primeira posição do vetor C_k.

Assim, ao final desta iteração, $P'[k]$ conterá o valor correto da posição i do vetor de mínimo prefixo do vetor B, e a variável *aponta* conterá o índice, em B, do primeiro elemento de C_k que corresponde à folha mais à esquerda de T_k.

Na última iteração, $C_k = B$ e $P'[\log p]$ contém o valor da posição i do vetor mínimo prefixo de B, e a variável *aponta* contém o índice do primeiro elemento de B.

Assim, o Procedimento *Constrói_P'* corretamente calcula o vetor P'.

□

Para a determinação de S', temos um teorema e prova similares.

Lema 2 *A execução do Procedimento Constrói_P' em cada processador leva tempo seqüencial* $O(n/p)$.

Prova. O número máximo de iterações é $\log p$. Em cada iteração, o valor de *aponta* é decrementado ou permanece o mesmo. O pior caso é quando $i = p - 1$. Neste caso, em cada iteração, o valor da variável *aponta* é atualizado. Como o número de elementos de B é p, o algoritmo atualiza o valor de *aponta* no máximo p vezes. Assim, o Procedimento *Constrói_P'* é executado em tempo seqüencial $O(p) = O(n/p)$.

□

O teorema a seguir resume os resultados obtidos.

Teorema 2 *O algoritmo Mínimo_Intervalar(CGM) resolve o problema de mínimo intervalar em tempo* $O(n/p)$ *usando* $O(1)$ *rodadas de comunicação e memória* $O(n/p)$.

Prova. O passo 1 é executado em tempo seqüencial $O(n/p)$, não necessita de comunicação e utiliza memória $O(n/p)$. O passo 2 é executado em tempo seqüencial $O(n/p)$, utiliza espaço $O(p)$ e necessita de uma rodada de comunicação onde é enviado um valor e são recebidos $p-1$ dados. Pelo teorema 2, o passo 3 é executado em tempo seqüencial $O(n/p)$, não precisa de comunicação e utiliza memória $O(p)$. Portanto, o Algoritmo *Mínimo_Intervalar(CGM)* resolve o problema de mínimo intervalar em tempo seqüencial $O(n/p)$, usando $O(1)$ rodadas de comunicação, onde são trocados no máximo $O(p)$ dados e utilizando memória local $O(n/p)$.

□

1.8.4 Processamento de Consultas

Nesta seção, mostraremos como usar a saída do Algoritmo *Mínimo_Intervalar(CGM)* para responder a consultas $O(n/p)$ em tempo constante. Uma consulta $MIN(i, j)$ é determinada como

se segue. Assumimos que i e j são conhecidos por todos os processadores e o resultado será dado pelo processador 0. Se a_i e a_j estão no mesmo processador, então $MIN(i,j)$ pode ser determinado pelo passo 1 do Algoritmo *Mínimo_Intervalar(CGM)*. Caso contrário, suponha que a_i e a_j estão em processadores distintos $\bar{i}$ e $\bar{j}$, respectivamente, com $\bar{i} < \bar{j}$. Seja $direita(\bar{i})$, o índice em A do elemento mais à direita no vetor $A_{\bar{i}}$, e $esquerda(\bar{j})$ o elemento mais à esquerda no vetor $A_{\bar{j}}$. Calculamos $MIN(i, direita(\bar{i}))$ e $MIN(esquerda(\bar{j}), j)$, usando o passo 1. Temos, então, dois casos:

1. Se $\bar{j} = \bar{i} + 1$ então $MIN(i,j) = \min\{MIN(i, direita(\bar{i}), MIN(esquerda(\bar{j}), j)\}$.
2. Se $\bar{i} + 1 < \bar{j}$, então calculamos $MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1)$, usando o passo 3 do algoritmo. Notemos que $MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1)$ corresponde a $\min\{b_{\bar{i}+1}, \ldots, b_{\bar{j}-1}\}$. Assim, $MIN(i,j) = \min\{MIN(i, direita(\bar{i})), MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1), MIN(esquerda(\bar{j}), j)\}$.

O valor de $MIN(direita(\bar{i}) + 1, esquerda(\bar{j}) - 1)$ é obtido usando o passo 3, como descrito a seguir. Cada processador calcula $w = \text{LCA}(b_{\bar{i}+1}, b_{\bar{j}-1})$ em tempo constante, então determina o nível l_w e determina v e u, os filhos esquerdo e direito de w, respectivamente. O processador $\bar{i}+1$ calcula $S'[l_w - 1]$ e envia este valor para o processador 0. O processador $\bar{j} - 1$ calcula $P'[l_w - 1]$ e envia este valor para o processador 0. O processador 0, finalmente, calcula o mínimo entre os mínimos recebidos. Em ambos os casos, o processador 0 recebe os mínimos dos processadores $\bar{i}$ e $\bar{j}$.

Notemos, finalmente, que a corretude do algoritmo *Mínimo_Intervalar(CGM)* resulta das observações apresentadas nesta seção.

1.9 Prefixo Comum Mais Longo

Dada uma cadeia $C = c_1 \ldots c_n$, queremos pré-processá-la de forma que o comprimento do prefixo comum mais longo de dois sufixos quaisquer de C possa ser determinado em tempo constante. Em outras palavras, dados dois sufixos $C_i = c_i c_{i+1} \ldots c_n$ e $C_j = c_j c_{j+1} \ldots c_n$ de C, queremos determinar o maior l tal que $c_i c_{i+1} \ldots c_{i+l} = c_j c_{j+1} \ldots c_{j+l}$. Pela descrição de Landau e Vishkin [42], este pré-processamento pode ser feito da seguinte forma:

1. Contruir uma árvore de sufixos para a cadeia C. Desta forma, qualquer consulta do prefixo comum mais longo é feita através de uma consulta ao LCA da árvore de sufixos. Assim,
2. temos que pré-processar a árvore de sufixos de forma que consultas LCA possam ser respondidas em tempo constante.

A construção da árvore de sufixos é feita em tempo linear (ver seção 1.6), bem como o préprocesamento para consultas LCA (ver seção 1.7). Com a estrutura de dados obtida, as consultas do comprimento do prefixo comum mais longo de dois sufixos é feita em tempo constante.

1.10 Implementação

Para obtermos os dados experimentais, os algoritmos foram implementados em linguagem C. As execuções das implementações foram feitas na máquina *Parsytec PowerXplorer* do LCPD do IME/USP (Laboratório de Computação Paralela e Distribuída do IME/USP). Esta máquina é composta de 16 nós interconectados por uma topologia de grade bidimensional, cada nó é formado por um par *PowerPC 601* e *Transputer T805*, que compartilham uma memória local de 32 *Mbytes*. Os nós são divididos em 4 partições de 4 nós cada. Cada partição é alocada para se rodar uma aplicação e permanece exclusiva para esta aplicação, não compartilhando os recursos do nó com qualquer outra aplicação ou usuário. As demais partições não alocadas ficam livres para outras aplicações ou usuários.

A máquina *Parsytec PowerXplorer* utiliza o PARIX que é uma extensão paralela para o sistema operacional UNIX e contém todo o ferramental necessário para o desenvolvimento de aplicações paralelas. As ferramentas rodam em um *host* e o código da aplicação paralela roda na máquina *Parsytec PowerXplorer*.

Em ambiente PARIX, o mesmo código é executado em todos os nós, sendo que cada nó contém uma cópia do código e executa-o até o fim. O comportamento de cada nó depende do conteúdo do código, da posição do nó e dos dados locais. As informações entre nós devem ser trocadas via mensagens, não existindo uma memória compartilhada global.

O PARIX oferece uma extensão para a linguagem C para desenvolvimento de aplicações paralelas. Entretanto, por questões de portabilidade, as implementações foram feitas em linguagem C, usando a *interface PowerPVM*, que é uma extensão da *interface* PVM3 para aplicações rodando em ambiente PARIX.

A *interface* PVM (*Parallel Virtual Machine*) é um pacote que permite a computação concorrente em uma rede heterogênea de computadores paralelos ou seqüenciais. PVM é formado de duas partes: um processo *daemon* - que roda em cada máquina ou nó - e uma biblioteca de rotinas para desenvolvimento de aplicações paralelas (rotinas de iniciação de processos, comunicação entre processadores, sincronização, etc.).

A *interface PowerPVM* não permite a utilização de redes heterogêneas. O desempenho das aplicações utilizando esta *interface* é próxima ao desempenho de aplicações utilizando somente PARIX.

Na obtenção dos dados experimentais, utilizamos instâncias onde a quantidade de dados trocados fosse máxima, de forma a obtermos os tempos absolutos de pior caso, em relação à comunicação. Consideramos que os dados já se encontravam corretamente distribuídos nos processadores. Com os tempos absolutos, pudemos obter os *speedups* dos algoritmos. Usamos o termo *speedup* como o quociente entre o tempo de execução seqüencial e o tempo de execução em paralelo. Nos nossos experimentos utilizamos o algoritmo seqüencial que roda em cada processador, rodando em apenas um processador, para obtermos o tempo de execução do algoritmo seqüencial.

Capítulo 2

Busca Unidimensional de Padrões

2.1 Introdução

Sejam uma cadeia texto $T = \tau_1 \ldots \tau_n$, de comprimento n, e uma cadeia padrão $P = \rho_1 \ldots \rho_m$, de comprimento m, formadas por símbolos em um alfabeto Σ dado. Dizemos que o padrão P *ocorre* em uma posição s , $1 \leq s \leq n - m + 1$, de T se $\tau_s = \rho_1, \tau_{s+1} = \rho_2, \ldots, \tau_{s+m-1} = \rho_m$. Dizemos ainda que *existe uma ocorrência* de P na posição s de T. O problema de *busca unidimensional de padrões* consiste em determinar todas as posições s em T, $1 \leq s \leq n - m + 1$, em que o padrão P ocorre. Este problema aparece em aplicações como edição de texto, visão computacional, reconhecimento de fala e biologia molecular [37].

Uma primeira idéia ingênua de se solucionar o problema é percorrer o texto T, testando em cada posição deste se o padrão P ocorre. Esta primeira tentativa leva tempo $\Theta((n - m + 1)m)$, no pior caso. Já o algoritmo de Rabin-Karp [38] utiliza a idéia de se considerar cada símbolo como um valor decimal e, através de manipulações aritméticas, reduzir o número de operações para se encontrar ocorrências do padrão no texto. Este algoritmo tem tempo de pior caso $\Theta((n - m + 1)m)$, mas seu tempo de caso médio é $O(n + m)$.

O algoritmo KMP (Knuth-Morris-Pratt) [41] roda em tempo linear no tamanho da entrada $(n + m)$. Este tempo é obtido pela redução no número de comparações entre posições do padrão e do texto, através de informações previamente obtidas, que descartam de antemão posições em que o padrão não pode ocorrer. Estas informações são obtidas ao se comparar o padrão com ele mesmo, em uma etapa de pré-processamento do padrão, onde uma função π é construída. Este algoritmo está descrito na seção a seguir e será utilizado como algoritmo básico para outros problemas de buscas de padrões. Outros algoritmos para este problema estão apresentados em [17, 30, 38, 42, 50].

Na busca unidimensional de padrões com escala, estamos interessados em determinar todas as ocorrências do padrão, escalado a k, no texto com $1 \leq k \leq \lfloor n/m \rfloor$. Amir *et al* [8], apresentaram um algoritmo de tempo linear no tamanho da entrada que determina todas as ocorrências escaladas. Este algoritmo utiliza o algoritmo KMP nas formas fatoradas das cadeias texto e

padrão.

Para ambos os problemas, e considerando p processadores, um texto de comprimento N e um padrão de comprimento m, com $m \leq N/p$, desenvolvemos algoritmos CGM que têm tempo de computação $O(N/p)$, consomem memória $O(N/p)$ e utilizam apenas uma rodada de comunicação em que são trocados no máximo $O(m)$ dados.

Na seção 2.2, apresentamos o algoritmo KMP baseado na versão descrita por Cormen *et al* [18]. Na seção 2.3, apresentamos a versão CGM para este problema que utiliza uma rodada de comunicação, onde são trocados $O(m)$ dados. Os resultados experimentais, para este caso, estão descritos na seção 2.3.4.

O algoritmo de Amir *et al*, para o caso de busca unidimensional com escala, é apresentado na seção 2.4. A versão CGM é apresentada na seção 2.5 e utiliza uma rodada de comunicação, onde são trocados $O(m)$ dados. Na seção 2.5.2, apresentamos os resultados experimentais obtidos com a implementação do algoritmo CGM.

Na seção 2.6, mencionamos ainda uma idéia de um algoritmo CGM para o caso sem escala que utiliza também uma rodada de comunicação, mas trocando apenas um dado. As idéias contidas nesta versão não podem, entretanto, ser estendidas para o caso com escala e para os casos bidimensionais. Por uma questão de uniformidade, mantivemos a versão onde são trocados no máximo $O(m)$ dados.

2.2 O Algoritmo KMP

Este algoritmo apresenta duas etapas: na etapa de pré-processamento do padrão, denominada análise do padrão, uma função π é construída. Na etapa de análise do texto, as ocorrências de P em T são obtidas. A versão aqui descrita é baseada em [18].

Denotaremos o prefixo $P = \rho_1 \dots \rho_k$ de k caracteres do padrão P por P_k. Assim, $P_0 = \varepsilon$ e $P_m = P = \rho_1 \dots \rho_m$. Com esta notação, o problema de busca unidimensional de padrões é encontrar todas as posições s em T, $1 \leq s \leq n - m + 1$, tais que $P \sqsupset T_{s+m-1}$.

2.2.1 Análise do Padrão

Esta etapa consiste em pré-processar o vetor P, comparando-o consigo mesmo, construindo-se, assim, uma função π, denominada *função prefixo*. Esta função irá direcionar a busca das possíveis ocorrências do padrão no texto. Dado um padrão $P = \rho_1 \dots \rho_m$, a função π para o padrão P é dada por:

$$\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m-1\}$$
$$\pi[q] = \max\{k : k < q \text{ e } P_k \sqsupset P_q\}.$$

Isto é, $\pi[q]$ é o comprimento do maior prefixo de P que é um sufixo próprio de P_q.

Exemplo 1

i	1	2	3	4	5	6	7	8	9	10
P_i	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

A seguir, descrevemos o procedimento para a análise do padrão. Os valores da função π serão armazenados em um vetor denominado também de π.

PROCEDIMENTO: Constrói_π
Entrada: o padrão P.
Saída: o vetor π de m posições.

1. $m \leftarrow comprimento(P)$
2. $\pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. **para** $q \leftarrow 2$ **até** m **faça**
5. **enquanto** $k > 0$ **e** $P[k+1] \neq P[q]$ **faça**
6. $k \leftarrow \pi[k]$
7. **se** $P[k+1] = P[q]$
8. $k \leftarrow k + 1$
9. $\pi[q] \leftarrow k$
10. **devolva** π

fim procedimento

Este procedimento leva tempo $O(m)$. As provas desta complexidade e da corretude do procedimento podem ser encontradas na seção 2.2.3.

2.2.2 Análise do Texto

O algoritmo de análise do texto é muito parecido com o procedimento de determinação da função prefixo, como podemos ver a seguir. A saída deste algoritmo será uma lista R, que armazena, ordenadamente, as posições de T onde P ocorre.

ALGORITMO: KMP
Entrada: o texto T e o padrão P.
Saída: uma lista R com as posições de T onde P ocorre.

1. $R \leftarrow \emptyset$
2. $n \leftarrow Comprimento(T)$
3. $m \leftarrow Comprimento(P)$
4. $\pi \leftarrow Constrói\_\pi(P)$
5. $q \leftarrow 0$
6. **para** $i \leftarrow 1$ **até** n **faça**

7. **enquanto** $q > 0$ **e** $P[q+1] \neq T[i]$ **faça**
8. $\quad q \leftarrow \pi[q]$
9. **se** $P[q+1] = T[i]$
10. $\quad q \leftarrow q+1$
11. **se** $q = m$
12. $\quad Insere(i-m+1, R)$
13. $\quad q \leftarrow \pi[q]$

fim algoritmo

Este algoritmo leva tempo $O(n+m)$, como veremos na seção a seguir, onde também demonstraremos sua corretude.

2.2.3 Tempo e Corretude

As demonstrações de tempo e corretude apresentadas nesta seção estão baseadas em [18]. Inicialmente, vamos demonstrar que o procedimento *Constrói_π* calcula o vetor π corretamente.

Definição 7 *Seja π uma função prefixo para um padrão P de m posições. Para cada posição q, $1 \leq q \leq m$, definimos:*

$$\pi^*[q] = \{\pi^0[q], \pi^1[q], \pi^2[q], \ldots, \pi^t[q]\},$$

onde π^j é definido como a função composta e, assim, temos $\pi^0[q] = q$ e $\pi^j[q] = \pi[\pi^{j-1}[q]]$, para todo $j > 1$. A seqüência pára quando $\pi^t[q] = 0$.

Definimos, ainda, $\pi^*[0] = \{0\}$.

No lema a seguir, mostramos que iterando a função prefixo π, podemos enumerar todos os prefixos P_k, que são sufixos de um prefixo P_q dado.

Lema 3 *Seja P um padrão de m posições e π o vetor correspondente à função π sobre P. Então, $\pi^*[q] = \{k : P_k \sqsupset P_q\}$, $q = 1, \ldots, m$.*

Prova. Se $i \in \pi^*[q]$, existe $w \in \{0, \ldots, t\}$ tal que $i = \pi^w[q]$. Vamos considerar $S = \{k : P_k \sqsupset P_q\}$, para algum q, $1 \leq q \leq m$.

Vamos mostrar, primeiramente, que se $i \in \pi^*[q]$ então $P_i \sqsupset P_q$ por indução em w.

A base da indução é quando $w = 0$. Neste caso $i = \pi^0[q] = q$ e, como $\sqsupset$ é uma relação, $P_q \sqsupset P_q$.

Temos que $i = \pi^w[q] = \pi[\pi^{w-1}[q]]$, pela definição 7. Fazendo $j = \pi^{w-1}[q]$, temos $i = \pi[j]$ e $P_i \sqsupset P_j$. Mas, por hipótese de indução, $P_j \sqsupset P_q$. Como $\sqsupset$ é transitiva, $P_i \sqsupset P_q$.

Com isso demonstramos que $\pi^*[q] \subseteq S$.

Vamos, agora, supor, por contradição, que exista $j \in S \setminus \pi^*[q]$, e vamos tomá-lo máximo. Notemos que $j < q$, pois $q \in S \cap \pi^*[q]$. Tomemos j' mínimo, $j < j' \leq q$ em $\pi^*[q]$. Temos, então, que $P_j \sqsupset P_q$, pois $j \in S$ e que $P_{j'} \sqsupset P_q$, pois $j' \in \pi^*[q]$. Como $j < j'$, $P_j \sqsupset P_{j'}$, mais ainda j é o maior valor com esta propriedade, então $\pi[j'] = j$ pela definição de π, e $j \in \pi^*[q]$, o que contradiz a escolha de j.

□

Lema 4 *Seja P um padrão de comprimento m e π a função prefixo calculada sobre P. Se $\pi[q] > 0$ então $\pi[q] - 1 \in \pi^*[q-1]$ para $q = 1, \ldots, m$.*

Prova. Seja $k = \pi[q]$. Como $k > 0$, temos que $P_k \sqsupset P_q$ e, conseqüentemente, descartando-se o último elemento de P_k e P_q, $P_{k-1} \sqsupset P_{q-1}$. Então, pelo lema anterior, $k - 1 \in \pi^*[q-1]$.

□

Definição 8 *Seja dado um padrão P de comprimento m. Definimos, para $q = 2, \ldots, m$, o subconjunto $E_{q-1} \subseteq \pi^*[q-1]$ por*

$$E_{q-1} = \{k : k \in \pi^*[q-1] \text{ e } P[k+1] = P[q]\}.$$

Este conjunto é formado pelos valores de k para os quais $P_k \sqsupset P_{q-1}$ e, além disso, por causa de $P[k+1] = P[q]$, $P_{k+1} \sqsupset P_q$.

Teorema 3 *Seja P um padrão de comprimento m e π a função prefixo calculada sobre P. Para $q = 2, \ldots, m$, temos*

$$\pi[q] = \begin{cases} 0 & \text{se} \quad E_{q-1} = \emptyset, \\ 1 + \max\{k \in E_{q-1}\} & \text{se} \quad E_{q-1} \neq \emptyset. \end{cases}$$

Prova. Se $r = \pi[q]$, então $P_r \sqsupset P_q$.

Se $r \geq 1$, pelo lema anterior, temos $P[r] = P[q]$. Assim, como $r = \pi[q]$, $r - 1 = \max\{k : P_k \sqsupset P_{q-1}\}$. Portanto, $r = 1 + \max\{k : P_k \sqsupset P_{q-1} \text{ e } P[k+1] = P[q]\} = E_{q-1}$ e $E_{q-1} \neq \emptyset$.

Se $r = 0$, vamos supor que exista t em E_{q-1}. Temos $t \in \pi^*[q-1]$ e $P[t+1] = P[q]$. Donde, $P_t \sqsupset P_{q-1}$ e $P[t+1] = P[q]$ e, por fim, $P_{t+1} \sqsupset P_q$ e $\pi[q] \geq t + 1 > 0$. Mas $\pi[q] = r = 0$, uma contradição. Logo $E_{q-1} = \emptyset$.

□

Teorema 4 *O procedimento Constrói_π calcula o vetor π corretamente.*

Prova. Para $q = 1$, na linha 2 temos $\pi[1] \leftarrow 0$ e, portanto, está correto, pois $\pi[q] < q$ para todo $q = 1, \ldots, m$.

Vamos mostrar, por indução, que o procedimento funciona corretamente. Vamos supor que $\pi[w]$ está correto para $w = 1, \ldots, q-1$.

No início de cada iteração, temos $k = \pi[q-1]$. Isto vale antes de se entrar pela primeira vez no laço e vale em cada iteração por causa da linha 9.

Nas linhas 5 e 6, calculamos k máximo em E_{q-1}, aplicando a hipótese de indução.

Quando saímos do laço da linha 5, temos duas opções:

- ou $P[k+1] = P[q]$ e $E_{q-1} \neq \emptyset$ e na linha 8 fazemos o incremento de k. Neste caso $\pi[q] = k+1$ pelo teorema 3, e portanto está correto.
- ou $k = 0$ e portanto $E_{q-1} = \emptyset$. O programa faz $\pi[q] \leftarrow k(=0)$ e, portanto, está correto.

□

Demonstraremos a corretude do algoritmo *KMP* a partir do lema a seguir.

Lema 5 *No fim de cada iteração do laço* ***para*** *da linha 6, temos que* $q \leq m$ *e* $P_q \sqsupset T_i$.

Prova. Se $m = 1$, para qualquer $i = 1, \ldots, n$ o algoritmo não executa a linha 8 e na linha 9 testa se $P[1] = T[i]$: em caso afirmativo $q \leftarrow 1$ e na linha 12 a posição i é inserida na lista R e $q \leftarrow 0$. Assim no final desta iteração temos $q < m$ e $P_q \sqsupset T_i$; caso contrário q continua com valor 0, as linhas 12 e 13 não são executadas e, portanto, $q < m$ e $P_q \sqsupset T_i$.

Vamos considerar o caso em que $m > 1$ e vamos provar, por indução, que as duas afirmações são válidas no fim de qualquer iteração $i = 1, \ldots, n$.

No início da primeira iteração, temos $i = 1$ e $q = 0$. A linha 8 não é executada, pois $q = 0$. Se, na linha 9, tivermos $P[1] = T[1]$, $q \leftarrow 1$ (linha 10) e esta iteração termina com $q = 1$ e $P_1 \sqsupset T_1$, pois $m > 1$ e as linhas 12 e 13 não são executadas. Caso contrário, se $P[1] \neq T[1]$, as linhas 9, 12 e 13 não são executadas e esta iteração termina com $q < m$ e $P_0 \sqsupset T_1$.

Agora, vamos supor que as afirmações sejam válidas no final das iterações $1, \ldots, i-1$ e vamos considerar q' o valor de q no fim da iteração $i-1$.

No início da iteração i teremos $q' < m$ e $P_{q'} \sqsupset T_{i-1}$, por hipótese de indução.

No final desta iteração, temos duas possibilidades:

- o valor de q' é decrementado pelas linhas 8 ou 13 e teremos $q < q' < m$. Portanto $q < m$; ou
- o valor de q' é incrementado de uma unidade pela linha 10, donde $q' \leq m$. Se $q' = m$, então a linha 13 é executada e logo $q < m$.

Para provarmos que $P_q \sqsupset T_i$, no final da iteração i, temos que considerar dois casos:

- $P[q'+1] \neq T[i]$. Neste caso, executa-se a linha 8 do laço **enquanto** da linha 7, decrementando-se o valor de q' até que o novo valor q seja igual a 0 ou que a igualdade $P[q+1] = T[i]$ seja atingida. Temos que o novo valor q está em $\pi^*[q']$ e $P_q \sqsupset P_{q'}$.

  - se $P[q+1] \neq T[i]$ e $q = 0$, q não é incrementado pela linha 10 e não são executadas as linhas 12 e 13. Logo, $P_0 \sqsupset T_i$.
  - se $P[q+1] = T[i]$ e $q = 0$, q é incrementado de uma unidade pela linha 10 e as linhas 12 e 13 não são executadas. Logo, $P_1 \sqsupset T_i$.
  - se $P[q+1] = T[i]$ e $q > 0$, e tendo que $P_q \sqsupset P_{q'}$, $P_{q'} \sqsupset T_{i-1}$ implica $P_q \sqsupset T_{i-1}$, obtemos $P_{q+1} \sqsupset T_i$. E como, na linha 10, $q \leftarrow q+1$, chegamos a $P_q \sqsupset T_i$.

- $P[q'+1] = T[i]$. Pela hipótese de indução, $P_{q'} \sqsupset T_{i-1}$. Como a linha 8 não é executada nesta iteração, a linha 9 incrementa q de uma unidade, isto é, $q = q'+1$. Ao final da linha 10 teremos $P[q] = T[i]$ e $P_{q'} \sqsupset T_{i-1}$. Portanto $P_q \sqsupset T_i$. Se as linhas 12 e 13 não forem executadas, continuamos tendo $P_q \sqsupset T_i$. Entretanto, se as linhas 12 e 13 forem executadas, o valor de q é decrementado para um valor q''. Como $P_{q''} \sqsupset P_q$ temos que $P_{q''} \sqsupset T_i$, onde q'' é o valor final de q.

Portanto, no final de uma iteração qualquer, as afirmações são válidas.

□

Teorema 5 *O algoritmo KMP corretamente calcula as posições de ocorrência de P em T.*

Prova. Vamos provar primeiro que se s está na lista R então $P \sqsupset T_{s+m-1}$.

Vamos supor que s seja incluído em uma iteração i, logo $s = i - m + 1$ e $i = s + m - 1$, pela linha 12. Pela linha 9, temos que $P[q] = T[i]$, pois do contrário $q < m$ e a linha 12 não é executada. Como, pelo lema anterior, $P_q \sqsupset T_i$ ao final da linha 10, temos que $P \sqsupset T_{s+m-1}$. Isto garante que nenhuma posição s é incorretamente colocada em R.

Mostraremos agora que se $P \sqsupset T_{s+m-1}$ então s pertence à lista R. Vamos supor que a hipótese seja válida mas $s \notin R$.

No início da iteração $i = s + m - 1$, seja $\bar{q}$ o valor de q no início desta iteração, $\bar{q} < m - 1$, pois a linha 12 não é executada nesta iteração. Temos que $P_{\bar{q}} \sqsupset T_{i-1}$ e, portanto, $P_{\bar{q}} \sqsupset P_{m-1}$, pois $P \sqsupset T_{s+m-1}$.

Seja s', $s < s' < s + m - 1$, a iteração de maior índice em que houve a última atribuição $q \leftarrow \pi[q]$, pela linha 8 ou pela linha 13. Tal s' existe pois $\bar{q} < m - 1$.

No fim desta iteração s' temos dois casos:

- se apenas a linha 13 foi executada, então no início desta iteração s', $q' = m - 1$ e $P[q'+1] = T[s']$ (ver figura 2.1(a) e (b)). Seja q'' o valor de q após executarmos a linha 13. Temos que $q'' < \bar{q} < m$, pois esta foi a última iteração em que houve decremento,

e portanto $q'' < s' - s + 1$ (ver figura 2.1 (b) e (d)). Como $P_{q''} = T[s' - q'' + 1 \ldots s'] = P[s' - s - q'' + 2 \ldots s' - s + 1]$ (ver figura 2.1 (b), (c) e (d)), temos que $P_{q''} \sqsupset P_{s'-s+1}$ (ver figura 2.1 (c) e (d)) e $P_{s'-s+1} \sqsupset P$ (ver figura 2.1 (a) e (c)), pois existe uma ocorrência de P em $s' - m + 1$. Uma contradição pois q'' é o valor máximo entre os comprimentos de prefixos de P que são sufixos de P. Portanto, o valor que q'' deve assumir é $s' - s + 1$ e s será inserido na lista R.

- se a linha 8 foi executada, seja q' o valor de q no início da iteração e seja q'' o valor de q ao fim do laço **enquanto** da linha 7. Temos $P[q'' + 1] = T[s']$ e $q'' < q' < m \Rightarrow q'' < s' - s + 1$. Analogamente, chegamos a uma contradição na determinação de q''.

Portanto se $P \sqsupset T_{s+m-1}$ então $s \in R$.

□

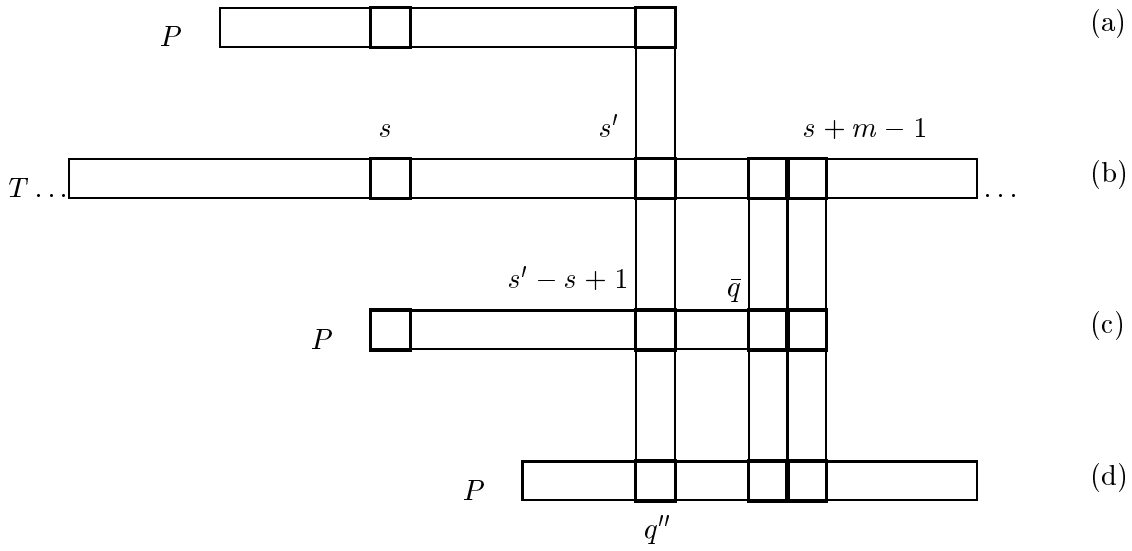

Figura 2.1: Figura para ilustrar um passo da demostração de $P \sqsupset T_{s+m-1} \Rightarrow s \in R$.

A complexidade de tempo do procedimento *Constrói_π* é dada pelo teorema a seguir.

Teorema 6 *O procedimento Constrói_π leva, no pior caso, tempo $O(m)$ para calcular o vetor π.*

Prova. Vamos fazer uma análise amortizada do algoritmo. Vamos associar um potencial k ao estado k corrente do algoritmo. Este potencial tem valor inicial igual a 0, pela linha 3. A linha 6

decrementa o valor de k sempre que é executada. Como $\pi[k] \geq 0$ para todo k, k nunca se torna negativo. A linha 8 é a única outra linha que afeta o valor de k, incrementando-o no máximo em uma unidade a cada vez que o laço **para** da linha 4 é executado. Como no início da primeira iteração $k < q$ e como q é incrementado em cada iteração, $k < q$ vale em todas as iterações.

Podemos pagar para cada execução do laço **enquanto** da linha 5 com o correspondente decremento na função potencial, pois $\pi[k] < k$. A linha 8 incrementa a função potencial em no máximo uma unidade. Assim, o custo amortizado do corpo do laço **para** da linha 5 é $O(1)$. Como o número de repetições deste laço é $O(m)$, temos que o tempo amortizado de pior caso é $O(m)$.

□

Teorema 7 *O algoritmo KMP leva, no pior caso, tempo $O(m+n)$ para determinar as ocorrências de P em T.*

Prova. O corpo do laço **para** da linha 6 tem tempo amortizado $O(1)$ por argumentos análogos aos envolvidos na demonstração do teorema anterior. Como este laço é repetido n vezes, temos que ele leva tempo $O(n)$. Pelo teorema anterior, a linha 4 leva tempo $O(m)$. Logo, o algoritmo tem tempo de pior caso $O(m + n)$.

□

2.3 O Algoritmo CGM de Busca Unidimensional sem Escala

Nesta seção, apresentaremos um algoritmo CGM, utilizando p processadores para o problema de busca unidimensional de padrões em um texto de comprimento N e um padrão de comprimento m, $m \leq N/p$, que é executado em tempo local $O(N/p)$, consome memória $O(N/p)$ e utiliza uma rodada de comunicação em que são trocados $O(m)$ dados. Vamos considerar $n = N/p$.

Assim como no algoritmo seqüencial KMP [41], estes algoritmos CGM compreendem uma fase de análise do padrão e uma fase de análise do texto. Esta última é a de determinação efetiva das posições do texto onde o padrão ocorre. Para este problema dois processadores serão ditos *vizinhos* se seus índices diferirem em uma unidade. Dado um processador de índice i, seu *processador vizinho esquerdo* será o processador de índice $i - 1$, para $1 \leq i \leq p - 1$; e o seu *processador vizinho direito* será o processador de índice $i + 1$, para $0 \leq i \leq p - 2$.

O texto inicialmente será dividido em p *subvetores de texto* S, também denominados *segmentos de texto*, cada um de tamanho $n = N/p$, que serão distribuídos nos processadores de forma que o primeiro e o último segmentos sejam armazenados nos processadores de índices 0 e $p - 1$, respectivamente, e que segmentos contíguos sejam armazenados em processadores com índices sucessivos, mantendo-se a contigüidade. Assim, dois processadores vizinhos conterão segmentos de texto adjacentes. O padrão será armazenado integralmente em cada processador. A Figura 2.2 ilustra a distribuição do texto entre os processadores.

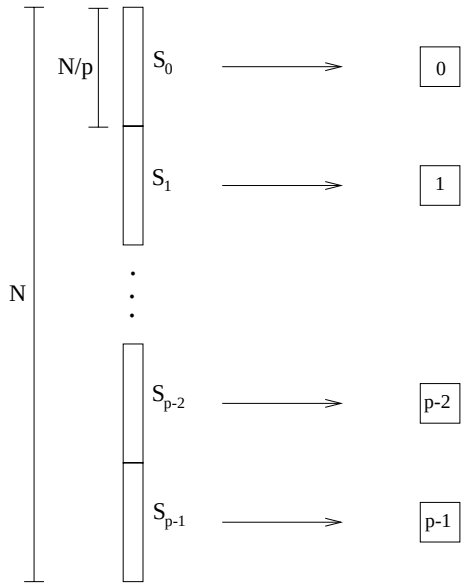

Figura 2.2: Esquema de distribuição de um texto de comprimento N entre p processadores.

Como o padrão deve ser comparado com o texto como se este estivesse contiguamente armazenado, devemos considerar o caso em que o padrão deve ser comparado com subvetores de texto armazenados em dois processadores vizinhos. Neste caso, diremos que o padrão estará sendo comparado na *fronteira* destes subvetores de texto. Desta forma, as fronteiras constituem-se de prefixos e sufixos de comprimento $m-1$ dos subvetores em cada processador, com exceção dos processadores que armazenam os subvetores das extremidades, que apresentam apenas uma fronteira. A fronteira correspondente ao prefixo de um subvetor de texto será denominado *fronteira esquerda* e, analogamente, a fronteira correspondente ao sufixo de um subvetor de texto será denominado *fronteira direita*.

2.3.1 Análise do Padrão

Na etapa de análise de padrão para o algoritmo no modelo CGM, além da função prefixo π descrita na seção 2.2.1, vamos definir uma *função sufixo* σ que está descrita a seguir:

$$\sigma : \{1, 2, \ldots, m\} \to \{2, \ldots, m+1\}$$
$$\sigma[q] = (m+1) - \max\{k : k < q \text{ e } P_{m-k+1\ldots m} \sqsubset P_{q\ldots m}\}.$$

Em outras palavras, $m - \sigma[q] + 1$ é o comprimento do maior sufixo de P que é um prefixo próprio de $P[q \ldots m]$.

Exemplo 2

i	1	2	3	4	5	6	7	8	9	10
P_i	a	c	b	a	b	a	b	a	b	a
$\sigma[i]$	10	11	5	6	7	8	9	10	11	11

Esta função funciona de maneira simétrica à função prefixo e servirá para a análise do texto, quando o padrão estiver sendo comparado na fronteira. Ela será calculada localmente em cada processador.

PROCEDIMENTO: Constrói_σ.
Entrada: o padrão P.
Saída: o vetor σ de m posições.
1. $m \leftarrow Comprimento(P)$
2. $\sigma[m] \leftarrow m + 1$
3. $k \leftarrow m + 1$
4. **para** $q \leftarrow m - 1$ **até** 1 **passo** -1 **faça**
5. **enquanto** $k < m + 1$ **e** $P[k - 1] \neq P[q]$ **faça**
6. $k \leftarrow \sigma[k]$
7. **se** $P[k - 1] = P[q]$
8. $k \leftarrow k - 1$
9. $\sigma[q] \leftarrow k$
10. **devolva** σ

fim procedimento

Desta forma, em cada processador, a etapa de análise do padrão consiste na construção dos vetores π e σ.

2.3.2 Análise do Texto

A saída deste algoritmo, em cada processador, será uma lista resposta R, de comprimento no máximo n, das posições do segmento de texto S onde P ocorre. A análise do texto envolve quatro passos principais executados em cada processador. O primeiro consiste em executar o algoritmo *KMP* localmente de forma a obter as ocorrências de P no subvetor de texto armazenado em cada processador. Após este passo, o vetor π já se encontra calculado e a possibilidade de ocorrência de P na fronteira direita é feita, construindo-se uma lista de candidatos da fronteira direita, que chamaremos C_D, com as posições que podem ser o início de uma ocorrência de P em S. Obtém-se então o vetor σ e com ele constrói-se uma lista de candidatos da fronteira esquerda, denominada C_E, que armazena as posições da fronteira esquerda que podem vir a ser a parte final de uma ocorrência de P em S no processador vizinho esquerdo. Finalmente, através das listas C_D e C_E obtêm-se as posições onde P efetivamente ocorre na fronteira. As listas C_D e C_E

têm comprimento no máximo m. No exemplo 3(a) temos uma cadeia texto T de comprimento 80 e uma cadeia padrão P de comprimento 7, distribuídas entre 4 processadores juntamente com os vetores π e σ calculados.

ALGORITMO: KMP_CGM
Entrada: um vetor T de texto, um vetor P de padrão.
Saída: listas R nos processadores com as posições do segmento local de texto S onde P ocorre.
{ Cada processador recebe $n = N/p$ posições contíguas do vetor T, repartido em subvetores $S_0, S_1, \ldots, S_{p-1}$ e o vetor P. Isto é, cada processador i recebe o subvetor S_i. }

1. Cada processador i executa o algoritmo *KMP* (página 25) para o segmento de texto S_i, obtendo a lista R.
2. Cada processador i, $i < p-1$, executa o procedimento *Fronteira_Direita*, obtendo a lista C_D.
3. Cada processador i, $i > 0$, executa o procedimento *Constrói_σ*.
4. Cada processador i, $i > 0$, executa o procedimento *Fronteira_Esquerda*, obtendo a lista C_E.
5. Cada processador i, $i > 0$, envia a lista C_E para o processador $i-1$.
6. Cada processador i, $i < p-1$, executa o procedimento *Combina*.

fim algoritmo

Exemplo 3(a) *Um exemplo de entrada de dados e sua distribuição entre $p = 4$ processadores, com os vetores π e σ calculados.*

Entrada:

$T = aaababababaaabbababababababbababababaabaababababaabbababbaababababaaababababaaaaba$

$P = abababa$

No processador 0:
$S_0 = aaababababaaabbababa$
$P = abababa$

No processador 1:
$S_1 = bababababbababababaaba$
$P = abababa$

No processador 2:
$S_2 = abababaabbababbaabab$
$P = abababa$

No processador 3:
$S_3 = abababaababababaaaaba$
$P = abababa$

Vetores π e σ em cada processador p_i, $0 \leq i \leq 3$:

i	1	2	3	4	5	6	7
π	0	0	1	2	3	4	5
σ	3	4	5	6	7	8	8

Os procedimentos *Fronteira_Direita* e *Fronteira_Esquerda* são responsáveis por determinar possíveis posições de ocorrência do padrão no texto que não foram analisadas no passo 1. Estas são as posições em que o padrão se encontra na fronteira, dividido entre o fim do texto em um processador e o início do texto no próximo processador. A seguir descreveremos os procedimentos *Fronteira_Direita* e *Fronteira_Esquerda*. A idéia do primeiro procedimento é executar o algoritmo *KMP* nas últimas $m-1$ posições do subvetor de texto e determinar as posições que são possíveis ocorrências do padrão mas que dependem dos dados contidos no processador vizinho direito para que sua qualidade de posição de ocorrência seja mantida. No segundo procedimento, percorrem-se as $m-1$ posições iniciais do subvetor de texto obtendo-se os candidatos que irão, ou não, certificar a condição de posição de ocorrência do subvetor no processador vizinho esquerdo. Na Figura 2.3, mostramos uma seqüência de segmentos contíguos de texto, suas regiões de fronteira e a direção em que são feitas as comunições entre vizinhos. No exemplo 3(b), temos as listas R, C_D e C_E calculadas em cada processador e como será o envio.

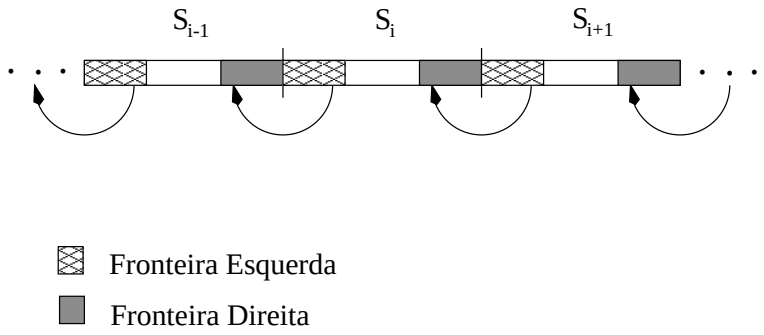

Figura 2.3: Fronteiras e comunicação entre vizinhos.

PROCEDIMENTO: Fronteira_Direita
Entrada: um segmento de texto S e o padrão P.
Saída: a lista C_D com as posições em que P pode ocorrer em S na fronteira direita.

1. $C_D \leftarrow \emptyset$
2. $n \leftarrow Comprimento(S)$
3. $m \leftarrow Comprimento(P)$
4. $q \leftarrow 0$
5. **para** $i \leftarrow n - m + 2$ **até** n **faça**
6. **enquanto** $q > 0$ **e** $P[q+1] \neq S[i]$ **faça**
7. $q \leftarrow \pi[q]$
8. **se** $P[q+1] = S[i]$
9. $q \leftarrow q + 1$
10. **enquanto** $q > 0$ **faça**
11. $Insere(n - q + 1, C_D)$
12. $q \leftarrow \pi[q]$
13. **devolva** C_D

fim procedimento

PROCEDIMENTO: Fronteira_Esquerda.
Entrada: um segmento de texto S e o padrão P.
Saída: a lista C_E com as posições que correspondem, na fronteira esquerda, ao fim de uma ocorrência de P em S no processador vizinho.

1. $C_E \leftarrow \emptyset$
2. $n \leftarrow Comprimento(S)$
3. $m \leftarrow Comprimento(P)$
4. $q \leftarrow m$
5. **para** $i \leftarrow m-1$ **até** 1 **passo** -1 **faça**
6. **enquanto** $q < m$ **e** $P[q-1] \neq S[i]$ **faça**
7. $q \leftarrow \sigma[q]$
8. **se** $P[q-1] = S[i]$
9. $q \leftarrow q-1$
10. **enquanto** $q < m$ **faça**
11. $Insere(m-q, C_E)$
12. $q \leftarrow \sigma[q]$
13. **devolva** C_E

fim procedimento

Exemplo 3(b) *Listas R, C_D e C_E calculadas em cada processador e envio para os vizinhos esquerdos.*

No processador 0:
$R = 3 \rightarrow 5$
$C_D = 16 \rightarrow 18 \rightarrow 20$
C_E *não é calculado.*

No processador 1:
$R = 9 \rightarrow 11$
$C_D = 18 \rightarrow 20$
$C_E = 2 \rightarrow 4 \rightarrow 6$

1 envia para 0:
$C_E = 2 \rightarrow 4 \rightarrow 6$

No processador 2:
$R = 1$
$C_D = 17 \rightarrow 19$
$C_E = 1 \rightarrow 3 \rightarrow 5$

2 envia para 1:
$C_E = 1 \rightarrow 3 \rightarrow 5$

No processador 3:
$R = 7 \rightarrow 9$
C_D *não é calculado.*
$C_E = 1 \rightarrow 3 \rightarrow 5$

3 envia para 2:
$C_E = 1 \rightarrow 3 \rightarrow 5$

Ambos procedimentos levam tempo $O(m)$, como veremos na seção 2.3.3

Como já foi dito, tendo construído as listas C_D, pelo procedimento *Fronteira_Direita*, e C_E, pelo procedimento *Fronteira_Esquerda* no processador sucessor, vamos combiná-las e assim certificar, ou não, as posições da fronteira direita armazenadas em C_D. Isto é feito pelo procedimento descrito a seguir. A lista de posições de confirmação recebida do vizinho direito será armazenada localmente na lista C_E.

PROCEDIMENTO: Combina
Entrada: as listas C_D, C_E e R.
Saída: a lista R com as posições que correspondem a ocorrências de P no segmento de texto S na fronteira direita incluídas.
1. $R' \leftarrow C_D$
2. $d \leftarrow Elemento(C_D)$
3. $e \leftarrow Elemento(C_E)$
4. **enquanto** $C_D \neq \emptyset$ **e** $C_E \neq \emptyset$ **faça**
5. **se** $n - d + e + 1 < m$
6. $Remove(C_E)$
7. $e \leftarrow Elemento(C_E)$
8. **caso contrário**
9. **se** $n - d + e + 1 > m$
10. $Remove(C_D)$
11. $d \leftarrow Elemento(C_D)$
12. **caso contrário**
13. $C_D \leftarrow Próximo(C_D)$
14. $d \leftarrow Elemento(C_D)$
15. $Remove(C_E)$
16. $e \leftarrow Elemento(C_E)$
17. **enquanto** $C_D \neq \emptyset$ **faça**
18. $Remove(C_D)$
19. **devolva** $R \cup R'$

fim procedimento

O procedimento consiste em percorrer-se as listas C_D e C_E como se estas fossem intercaladas, mas ao invés de comparar seus valores, as distâncias das posições até o fim e até o começo de S, respectivamente, são adicionadas e comparadas com o tamanho do padrão. Então, dependendo desta comparação, caminha-se nas listas, removendo-se elementos da lista C_E e mantendo-se ou removendo-se elementos da lista C_D, cujo primeiro elemento se encontra apontado por R'. Ao final, se C_D ainda contiver algum elemento, estes serão simplesmente removidos. O procedimento devolve a lista R concatenada com a lista R', contendo todos os inícios de ocorrências do padrão P no segmento de texto S armazenado localmente. O exemplo 3(c) contém os dados calculados, os dados recebidos e as listas R resultantes em cada processador após a aplicação do procedimento *Combina*.

Exemplo 3(c) *Recebimento dos dados e obtenção da lista R final.*

Processador 0 recebe:
$C_E = 2 \rightarrow 4 \rightarrow 6$
e calculou:
$C_D = 16 \rightarrow 18 \rightarrow 20$
$R = 3 \rightarrow 5$

Processador 1 recebe:
$C_E = 1 \rightarrow 3 \rightarrow 5$
e calculou:
$C_D = 18 \rightarrow 20$
$R = 9 \rightarrow 11$

Processador 2 recebe:
$C_E = 1 \rightarrow 3 \rightarrow 5$
e calculou:
$C_D = 17 \rightarrow 19$
$R = 1$

Processador 3 não recebe dados e calculou:
$R = 7 \rightarrow 9$

Percorremos as listas C_D *e* C_E *verificando o valor de* $n - Elemento(C_D) + Elemento(C_E) + 1$. *Neste exemplo* $n = 20$.

- *No processador 0.*
  Como $20 - 16 + 2 + 1 = 7$, $20 - 18 + 4 + 1 = 7$ *e* $20 - 20 + 6 + 1 = 7$, *os valores 16, 18 e 20 são incluídos na lista R. Assim,* $R = 3 \rightarrow 5 \rightarrow 16 \rightarrow 18 \rightarrow 20$.

- *No processador 1.*
  Como $20 - 18 + 1 + 1 < 7$, *1 é removido de* C_E. *Como* $20 - 18 + 3 + 1 < 7$, *3 é removido de* C_E. *Como* $20 - 18 + 5 + 1 > 7$, *18 é removido de* C_D. *Como* $20 - 20 + 5 + 1 < 7$, *5 é removido de* C_E. *Finalmente, como* C_D *está vazia,* C_D *é então esvaziada e nenhum elemento é inserido na lista R. Assim,* $R = 9 \rightarrow 11$.

- *No processador 2.*
  Como $20 - 17 + 1 + 1 < 7$, *1 é removido da lista* C_E. *Como* $20 - 17 + 3 + 1 = 7$ *e* $20 - 19 + 5 + 1 = 7$, *os valores 17 e 19 são incluídos na lista R. Assim,* $R = 1 \rightarrow 17 \rightarrow 19$.

- *O processador 3 não recebe dados e assim é apenas mantida a lista* $R = 7 \rightarrow 9$.

2.3.3 Tempo e Corretude

Para analisar o tempo de pior caso e a corretude do algoritmo *KMP_CGM*, vamos verificar o tempo de pior caso e a corretude de cada procedimento envolvido na descrição deste.

A corretude do procedimento *Constrói_σ* tem demonstração análoga à corretude do procedimento *Constrói_π* (página 25). A complexidade de pior caso também é $O(m)$ por motivos análogos.

Teorema 8 *Os procedimentos Fronteira_Direita e Fronteira_Esquerda têm tempo de pior caso $O(m)$, cada um.*

Prova. Em ambos os casos, o corpo do laço **para** da linha 5 é executado em tempo amortizado $O(1)$. Como o laço é executado $m-1$ vezes, este laço tem tempo de pior caso $O(m)$.

Como, no primeiro procedimento, q é incrementado em no máximo uma unidade em cada iteração, ao final do laço **para** da linha 5, seu valor será menor que m. O laço **enquanto** da linha 10 decrementa o valor de q, com valor inicial $q < m$, até que este seja igual a 0. Logo este primeiro procedimento leva tempo $O(m)$.

No segundo procedimento, ao final do laço **para** da linha 5, seu valor será no mínimo 1. O laço **enquanto** da linha 10 incrementa o valor de q, partindo de $q \geq 1$, até que este seja igual a m. Logo, este procedimento também leva tempo $O(m)$.

□

Os procedimentos *Fronteira_Direita* e *Fronteira_Esquerda* têm as demonstrações de corretude análogas. Assim, iremos enunciar os resultados para o primeiro. Vamos, primeiramente, enunciar um lema e em seguida o teorema de corretude.

Lema 6 *O invariante $P_q \sqsupset S$ é válido em qualquer iteração do laço **enquanto** da linha 10 do procedimento Fronteira_Direita.*

Prova. No final do laço **para** da linha 5, pelo Lema 5, vale $P_q \sqsupset S$. Logo, também vale no início da primeira iteração do laço **enquanto** da linha 10.

Em qualquer iteração, temos na linha 12 $q \leftarrow \pi[q]$. Seja q' o novo valor de q após a linha 12. Pela definição de π, $P_{q'} \sqsupset P_q$, mas como $P_q \sqsupset S$, temos que no final de qualquer iteração do laço **enquanto** da linha 10, $P_{q'} \sqsupset S$.

□

Teorema 9 *O procedimento Fronteira_Direita calcula corretamente as possíveis ocorrências de P em S na fronteira direita.*

Prova. Vamos mostrar que $s \in C_D$ se, e somente se, $P_{n-s+1} \sqsupset S$.

Na ida, vamos considerar i uma iteração qualquer do laço **enquanto** da linha 10 quando s é inserido em C_D. Como $s = n - q + 1$, $q > 0$, pela linha 12 e pelo lema anterior, $P_q \sqsupset S$, temos então que $q = n - s + 1$ e $P_{n-s+1} \sqsupset S$.

Na volta, vamos supor que $s \notin C_D$ e $P_{n-s+1} \sqsupset S$ e s mínimo.

Como $s \notin C_D$, $q = 0$. Seja $s' = n - q + 1$ o último valor inserido em C_D, então $P_{n-s'+1} \sqsupset S$ e $q = 0$ no final desta iteração.

Na linha 12, $q \leftarrow \pi[n - s' + 1]$ e $\pi[n - s' + 1] = 0$, pois $q = 0$ ao final desta iteração.

Mas $P_{n-s+1} \sqsupset P_{n-s'+1}$, uma contradição na obtenção do valor final de q.

□

As demonstrações do tempo e da corretude do procedimento *Combina* se encontram nos teoremas a seguir.

Teorema 10 *O procedimento Combina tem tempo de pior caso $O(m)$.*

Prova. Em cada iteração do laço **enquanto** da linha 4 um elemento de C_E é removido ou um elemento de C_D é removido ou caminha-se na lista C_D e remove-se um elemento de C_E. Como C_D e C_E têm no máximo $m - 1$ elementos cada um e o corpo do laço leva tempo $O(1)$ para ser executado, este laço leva tempo máximo $O(m)$.

O laço **enquanto** da linha 17 também leva tempo $O(m)$ pois em cada iteração um elemento de C_D é removido.

Portanto, o tempo de pior caso do procedimento é $O(m)$.

□

Teorema 11 *O procedimento Combina calcula corretamente as posições onde uma ocorrência de P em T efetivamente ocorre.*

Prova. O laço **enquanto** da linha 4 pára pois C_D e C_E são listas finitas e a cada iteração é removido um elemento de C_D ou um elemento de C_E.

Para mostrar que o procedimento calcula corretamente as ocorrências, devemos mostrar que $s \in R'$ se, e somente se, $d \in C_D$, $e \in C_E$, $P_{n-d+1} \sqsupset S$, $P_e \sqsubset S$ e $n - d + e + 1 = m$, com $d = s$.

Inicialmente, se $s \in R'$ então $s \in C_D$ não foi removido de C_D. Assim, nem as linhas 10-11, nem a linha 18 foram executadas com $d = s$.

Como $d = s$, o laço **enquanto** da linha 4 é executado percorrendo-se a lista C_E. O laço não pára com $C_E = \emptyset$ pois $C_D \neq \emptyset$ e s seria removido pela linha 18. Como as linhas 10-11 também não são executadas com $d = s$ e C_E tem tamanho finito, em algum instante as linhas 13-16 são executadas e $n - d + e + 1 = m$, com $P_{n-d+1} \sqsupset S$ pois $d \in C_D$, e $P_e \sqsubset S$ pois $e \in C_E$.

Vamos supor, agora, que $s \notin R'$ mas $s \in C_D$. Seja i a primeira iteração que começa com $d = s$ e seja e' o valor de e.

Como $s \notin R'$ então em algum instante s foi removido de C_D. Temos dois casos:

- as linhas 10 e 11 foram executadas com $d = s$ então $n - d + e + 1 > m$. Contradição pois $n - d + e + 1 = m$
- a linha 18 foi executada então, para todo $e > e'$, $e \in C_E$, $n - d + e + 1 < m$. Contradição.

Logo, $s \in R'$.

Como o valor de R, pelo Teorema 5, está corretamente calculado, $R \cup R'$ contém os valores corretos das posições de S onde P ocorre.

□

Com os resultados obtidos, podemos então mostrar o tempo e a corretude do algoritmo *KMP_CGM*.

Teorema 12 *O algoritmo KMP_CGM leva tempo de computação local $O(N/p)$, usando memória $O(N/p)$ e $O(1)$ rodadas de comunicação em que são trocados $O(N/p)$ dados.*

Prova. O passo 1 leva tempo seqüencial $O(\frac{N}{p} + m)$ e obtém a lista R. Os passos 2, 3 e 4 levam tempo $O(m)$ cada um. Até aqui não é necessária nenhuma comunicação. No passo 5, a lista C_E, de comprimento $O(m)$ é enviada em uma rodada de comunicação. Finalmente, o passo 6 leva tempo $O(m)$ para obter o resultado final.

Em cada processador estarão armazenados: o segmento de texto de tamanho $O(N/p)$, o padrão de tamanho $O(m)$ e as listas C_D, C_E e R de tamanho máximo $O(m)$ cada uma, onde $m = O(N/p)$. Assim, a memória utilizada é $O(N/p)$.

□

Teorema 13 *O algoritmo KMP_CGM obtém corretamente as ocorrências de P em S_i em cada processador i, $0 \leq i < p$.*

Prova. Esta corretude decorre diretamente dos resultados obtidos nos teoremas anteriores.

□

Com isto, mostramos que o algoritmo *KMP_CGM* calcula corretamente as ocorrências de P no segmento de texto armazenado no processador, manipulando os dados em tempo seqüencial $O(\frac{N}{p} + m) \leq O(N/p)$ em uma rodada de comunicação quando são enviados no máximo $m = O(N/p)$ dados, e utilizando memória $O(N/p)$.

2.3.4 Resultados da Implementação

Neste caso, utilizamos cadeias de caracteres para o texto e para o padrão onde todos seus elementos são iguais ao símbolo a, pois este é um caso que demanda mais tempo. O algoritmo

CGM é executado para dois exemplos, em ambos utilizamos uma cadeia texto de comprimento $N = 2^{18}$ e duas cadeias padrão de comprimentos $m = 2^4$ e $m = 2^{14}$. As ocorrências aparecem em todas as posições da cadeia texto a menos das posições começando nas últimas $m - 1$ posições.

Para os testes, utilizamos 1, 2, 4, 8 e 16 processadores, obtendo os tempos absolutos mostrados no gráfico da Figura 2.4. No gráfico, podemos observar que para o padrão de comprimento $m = 2^4$, o tempo cai sensivelmente com o aumento do número de processadores. Isto porque a quantidade de dados trocados é pequena e o tempo de comunicação acaba sendo uma parcela bem menor do que a parcela do tempo de computação local. Para o padrão de comprimento $m = 2^{14}$, a diminuição do tempo é mais discreta, uma vez que a quantidade de dados trocados é maior. Os valores dos tempos absolutos, bem como dos *speedups* estão apresentados no apêndice A.

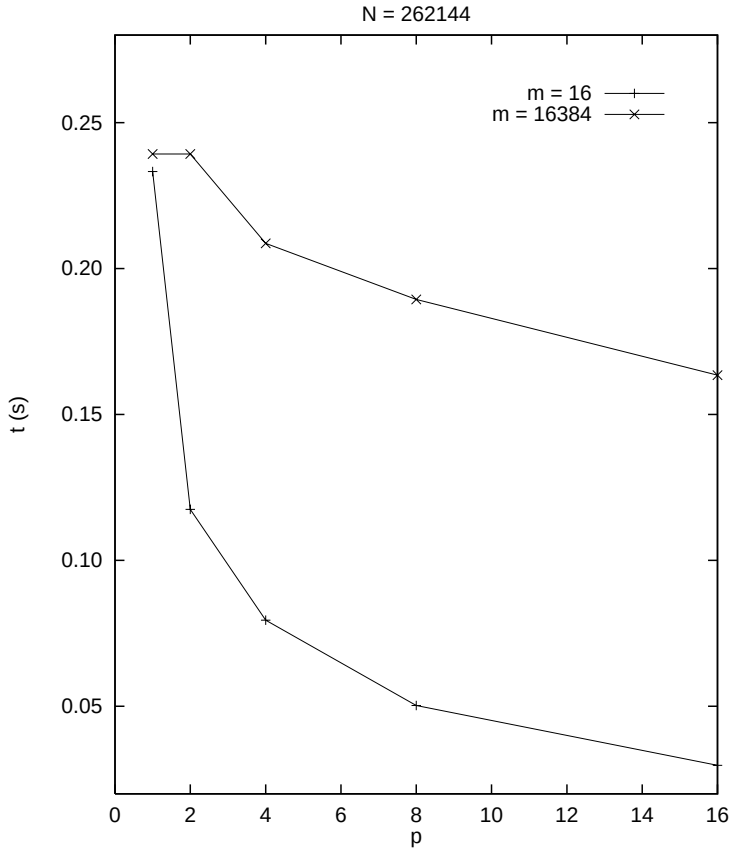

Figura 2.4: Tempo em s para $N = 2^{18} = 262144$, $m = 2^4 = 16$ e $m = 2^{14} = 16384$, utilizando 1, 2, 4, 8 e 16 processadores.

No gráfico da Figura 2.5, podemos acompanhar o *speedup* para os dois valores do comprimento do padrão. A linha tracejada marcada com triângulos cheios corresponde ao *speedup* ótimo. Vemos que o *speedup* é pequeno para o caso $m = 2^{14}$. Como o programa seqüencial é bastante rápido, a comunicação acaba dominando o tempo de computação local. Este fato é

confirmado pelo desempenho melhor para o padrão de comprimento $m = 2^4$.

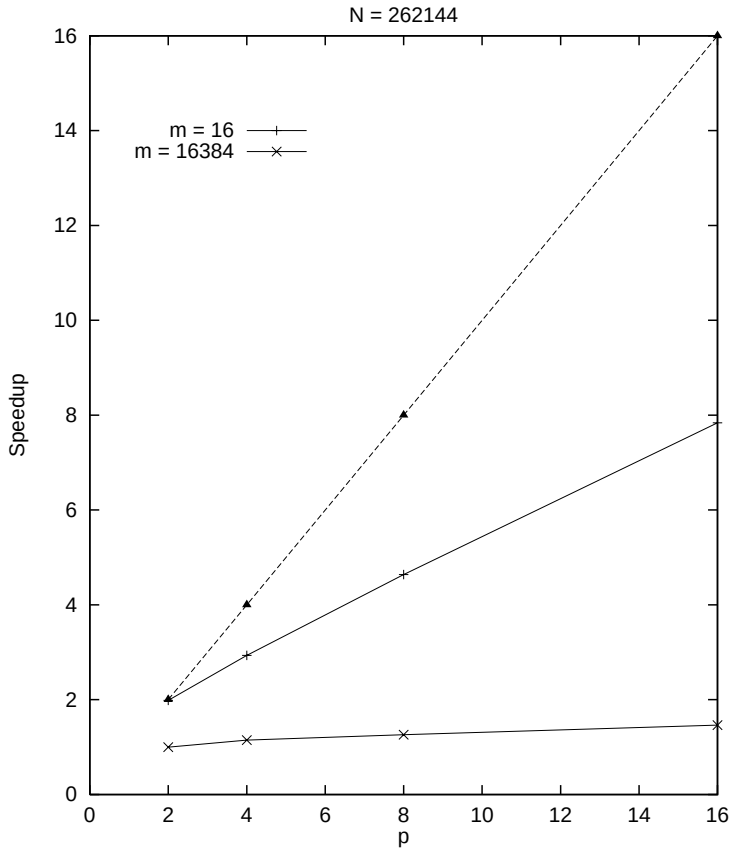

Figura 2.5: *Speedup* para 2, 4, 8 e 16 processadores com $N = 2^{18} = 262144$, $m = 2^4 = 16$ e $m = 2^{14} = 16384$.

2.4 Busca Unidimensional de Padrões com Escala

O problema de *busca unidimensional de padrões com escala* é uma generalização do problema de busca de padrões. Tendo como entrada um padrão $P = \rho_1 \cdots \rho_m$ e um texto $T = \tau_1 \cdots \tau_n$ com $n > m$, determinamos todas as posições de T onde uma ocorrência de P escalado a k (uma k-ocorrência) começa, para todo $k = 1, \ldots, \lfloor n/m \rfloor$. As k-ocorrências com $k = 1$ são aquelas da busca de padrões normal. O algoritmo descrito nesta seção foi desenvolvido por Amir *et al* [8].

Uma primeira idéia é a de se fazer uma busca de padrões para cada escala k, o que leva a um algoritmo de complexidade $O(n^2)$. Descreveremos um algoritmo que obtém todas as k-ocorrências do padrão no texto, com $1 \leq k \leq n/m$, em tempo linear no tamanho da entrada. Para tanto, vamos introduzir alguns conceitos e propriedades envolvidos na descrição do algoritmo.

Definição 9 *Seja $S = \sigma_1\sigma_2\cdots\sigma_n$ uma cadeia sobre um alfabeto Σ. A **representação fatorada** da cadeia S é a cadeia $\sigma_1'^{r_1}\sigma_2'^{r_2}\cdots\sigma_{\hat{n}}'^{r_{\hat{n}}}$, $\hat{n} \leq n$, tal que:*

1. *$\sigma_i' \neq \sigma_{i+1}'$, para $1 \leq i < \hat{n}$; e*
2. *S pode ser descrita como uma concatenação do símbolo σ_1' repetido r_1 vezes, o símbolo σ_2' repetido r_2 vezes, ..., e o símbolo $\sigma_{\hat{n}}'$ repetido $r_{\hat{n}}$ vezes.*

Denotamos por $S^\Sigma = \sigma_1'\sigma_2'\cdots\sigma_{\hat{n}}'$, a *parte de símbolos* de S e por $S^\#$ o vetor de números naturais $r_1, r_2, \ldots, r_{\hat{n}}$, a *parte de expoentes* de S.

Exemplo. Para $S = aabbbccabccbbaaaa$, $S' = a^2b^3c^2a^1b^1c^2b^2a^4$, $S^\Sigma = abcabcba$, e $S^\# = [2,3,2,1,1,2,2,4]$.

Observação 1 *Sejam T^Σ e $T^\# = [t_1, \ldots, t_{\hat{n}}]$ as partes de símbolos e expoentes, respectivamente, do vetor de texto T. Sejam P^Σ e $P^\# = [p_1, \ldots, p_{\hat{m}}]$ as partes de símbolos e expoentes, respectivamente, do vetor de padrão P. A determinação de todas as ocorrências de P escalado a k em T é equivalente à determinação de todas as posições i satisfazendo as condições a seguir:*

A. Existe uma ocorrência de P^Σ na posição i de T^Σ.

B.1. $t_i \geq kp_1$.

B.2. $t_{i+1} = kp_2, \ldots, t_{i+\hat{m}-2} = kp_{\hat{m}-1}$.

B.3. $t_{i+\hat{m}-1} \geq kp_{\hat{m}}$.

Observação 2 *Dadas as partes de expoentes $T^\#$ e $P^\#$ dos respectivos vetores T e P, podemos construir as cadeias $T' = t_2/t_1, t_3/t_2, \ldots, t_{\hat{n}}/t_{\hat{n}-1}$ de $\hat{n}-1$ elementos reais e $P' = p_3/p_2, p_4/p_3, \ldots, p_{\hat{m}-1}/p_{\hat{m}-2}$ de $\hat{m}-3$ elementos reais. Supondo $\hat{m} > 3$, a condição B.2 é satisfeita para $k = t_{i+1}/p_2$ se e somente se existe uma ocorrência da cadeia P' na posição $i+1$ da cadeia T'.*

O algoritmo tem como entrada os vetores T e P, mas trabalha com as partes de símbolos e de expoentes da representação fatorada, o que garante o tempo linear no tamanho da entrada. A determinação das k-ocorrências é baseada na observação anterior. Descreveremos o algoritmo supondo que $\hat{m} > 3$, os casos em que $\hat{m} \leq 3$ são trivialmente obtidos dentro da mesma complexidade de tempo. A saída do algoritmo será uma lista R que armazena os valores da posição e de k que correspondem à k-ocorrência de P em T.

ALGORITMO: Busca_com_Escala
Entrada: o texto T e o padrão P.
Saída: uma lista R com as posições de T onde existem k-ocorrências de P.
1. $R \leftarrow \emptyset$

2. $n \leftarrow Comprimento(T)$
3. $m \leftarrow Comprimento(P)$
4. $Constrói\_escala(T, T^{\Sigma}, T^{\#})$
5. $Constrói\_escala(P, P^{\Sigma}, P^{\#})$
6. $Constrói\_quociente(T^{\#})$
7. $Constrói\_quociente(P^{\#})$
8. $R_1 \leftarrow KMP(T^{\Sigma}, P^{\Sigma})$
9. $R_2 \leftarrow KMP(T', P')$
10. $pos \leftarrow 1$
11. $i \leftarrow Elemento(R_1)$
12. $j \leftarrow Elemento(R_2)$
13. **enquanto** $R_1 \neq \emptyset$ **e** $R_2 \neq \emptyset$ **faça**
14. **se** $j < i + 1$
15. $Remove(R_2)$
16. $j \leftarrow Elemento(R_2)$
17. **caso contrário**
18. **se** $j > i + 1$
19. $Remove(R_1)$
20. $i \leftarrow Elemento(R_1)$
21. **caso contrário**
22. $k \leftarrow t[i + 1]/p[2]$
23. **se** $t[i] \geq kp_1$ **e** $t[i + \hat{m} - 1] \geq kp[\hat{m}]$
24. $Insere((pos + (t[i] - kp[1]); k), R)$
25. $Remove(R_1)$
26. $i \leftarrow Elemento(R_1)$
27. $Remove(R_2)$
28. $j \leftarrow Elemento(R_2)$
29. $pos \leftarrow pos + t[i]$
30. **devolva** R

fim algoritmo

O procedimento *Constrói_escala* percorre as cadeias T e P construindo os vetores T^{Σ} e $T^{\#}$, e P^{Σ} e $P^{\#}$, com comprimentos $\hat{n}$ e $\hat{m}$, respectivamente.

O procedimento *Constrói_quociente* percorre os vetores $T^{\#}$ e $P^{\#}$ construindo os vetores de quocientes T' e P'. Ao final deste procedimento, o vetor T' terá $\hat{n} - 1$ posições e P' terá $\hat{m} - 3$ posições.

Após estas construções iniciais, o algoritmo aplica uma busca de padrões de P^{Σ} em T^{Σ}. O resultado deste procedimento estará armazenado em R_1, que é uma lista das posições de T^{Σ} onde uma ocorrência de P^{Σ} existe, de acordo com a condição A da observação 1. Com isto, temos uma primeira indicação de posições de k-ocorrências, que serão descartadas ou aceitas conforme os dados obtidos no passo seguinte.

O passo a seguir consiste em aplicar o algoritmo *KMP*, buscando ocorrências de P' em T'.

O resultado desta busca é armazenado em uma lista R_2 das posições onde uma ocorrência de P' em T' existe. Observemos que, na construção de P', os quocientes obtidos são $p_3/p_2, p_4/p_3, \ldots,$ $p_{\hat{m}-1}/p_{\hat{m}-2}$. Com isto, as posições em R_2 que armazenam ocorrências de P' em T', indicam posições de $T^{\#}$ onde ocorre $P^{\#}_{2\ldots\hat{m}-1}$, para alguma escala k, $1 \leq k \leq n/m$, satisfazendo, assim, a condição B.2 da observação 1 e acordando com a observação 2. Devemos observar também que P' armazena informações sobre $P^{\#}$ nas posições entre 2 e $\hat{m}-1$, inclusive estas.

Basta, agora, combinar as informações de R_1 e R_2 para se obter a lista R de resposta. Nesta etapa, para cada ocorrência de P^{Σ} em T^{Σ} em uma posição i, verifica-se se existe uma ocorrência de P' em T', na posição $i+1$. Se isto ocorrer, as condições A e B.2 da observação 1 estão satisfeitas, bastando testar as condições B.1 e B.3. Para isto, determina-se a escala k e verifica-se se as condições são satisfeitas. Em caso afirmativo, insere-se o valor da posição das k-ocorrências em T e o valor de k em R.

2.4.1 Tempo e Corretude

A corretude do algoritmo *Busca_com_Escala* decorre diretamente das observações 1 e 2. O tempo de pior caso é $O(n+m)$ e sua demonstração se encontra no teorema a seguir.

Teorema 14 *O algoritmo Busca_com_Escala tem tempo de pior caso* $O(n+m)$.

Prova. As linhas 4 e 6 levam tempo $O(n)$ cada. As linhas 5 e 7 levam tempo $O(m)$ cada. No pior caso, as linhas 8 e 9 levam tempo $O(n+m)$ cada e geram listas R_1 e R_2 de tamanho $O(n)$ cada.

Em cada passo do laço **enquanto** da linha 13 um elemento de R_1 e/ou um elemento de R_2 é removido. Como o corpo do laço leva tempo $O(1)$ e o laço pára quando uma das listas fica vazia, este laço leva tempo $O(n)$.

Assim, o algoritmo tem tempo de pior caso $O(n+m)$.

□

Este resultado mostra que o algoritmo determina todas as k-ocorrências, com $1 \leq k \leq n/m$, de P em T, em tempo de pior caso $O(n+m)$, que é o mesmo tempo do algoritmo *KMP*.

2.5 O Algoritmo CGM para Busca de Padrões com Escala

Nesta seção, apresentaremos um algoritmo CGM para o problema de busca de padrões unidimensional com escala. O algoritmo aqui apresentado está baseado nos algoritmos apresentados nas seções 2.2 e 2.3, com algumas adaptações e extensões.

Vamos considerar como entrada do algoritmo um texto T de tamanho N e um padrão P de tamanho m, ambos sobre um alfabeto Σ e que $N \gg m$, mais especificamente, $m \leq N/p$. Vamos considerar $n = N/p$.

Inicialmente, o texto será distribuído e armazenado nos processadores da mesma forma que o algoritmo CGM para a busca de padrões sem escala. O padrão será armazenado, por inteiro, localmente em cada processador. Estamos interessados em determinar as k-ocorrências de P em cada segmento de texto S, com $1 \leq k \leq N/(mp)$. Aqui, limitamos o intervalo de possíveis valores de k, de forma que P escalado ao valor máximo de k tenha comprimento igual ao tamanho do segmento de texto armazenado em cada processador.

No algoritmo CGM, os segmentos de texto e o padrão em cada processador são processados localmente de forma a se obter os vetores S^{Σ} e $S^{\#}$, e P^{Σ} e $P^{\#}$, e os vetores de quocientes S' e P'.

As k-ocorrências também podem existir nas regiões de fronteira como no caso sem escala. Neste caso, consideramos as fronteiras em termos dos segmentos S^{Σ} armazenados em cada processador e as fronteiras serão os sufixos e/ou prefixos de S^{Σ} de comprimento $\hat{m}$.

O algoritmo CGM para este problema funciona de maneira muito similar ao algoritmo CGM para o caso sem escala. As principais diferenças estão na execução local do algoritmo *Busca_com_Escala* (página 44) e alguns detalhes nas manipulações das fronteiras, que envolvem o envio de outros dados além da lista de candidatos.

ALGORITMO: Busca_com_Escala_CGM
Entrada: um vetor T de texto e um vetor P de padrão.
Saída: listas R nos processadores com as posições do segmento de texto S onde existem k-ocorrências de P.
{ Cada processador recebe $\frac{N}{p}$ posições contíguas do vetor T, repartido em subvetores $S_0, S_1, \ldots, S_{p-1}$ e o vetor P. }

1. Cada processador i executa o algoritmo *Busca_com_Escala* (página 44) para o segmento de texto S_i, obtendo a lista R, e os vetores S^{Σ}, $S^{\#}$, P^{Σ} e $P^{\#}$.
2. Cada processador i, $i < p-1$, executa o procedimento *Fronteira_Direita_escala*, obtendo a lista C_D.
3. Cada processador i, $i > 0$, executa o procedimento *Constrói_σ*.
4. Cada processador i, $i > 0$, executa o procedimento *Fronteira_Esquerda_escala*, obtendo a lista C_E.
5. Cada processador i, $i > 0$, envia a lista C_E, e os dados $S^{\Sigma}[1]$, $S^{\Sigma}[2]$, $S^{\#}[1]$ e $S^{\#}[2]$ para o processador $i-1$.
6. Cada processador i, $i < p-1$, executa o procedimento *Combina_escala*.

fim algoritmo

No exemplo 4(a) temos segmentos S distribuídos entre os processadores e o padrão P com os respectivos vetores S^{Σ}, $S^{\#}$ e S', P^{Σ}, $P^{\#}$ e P' calculados pelo passo 1. A lista R obtida em cada processador, para este exemplo se encontra no exemplo 4(b).

Exemplo 4(a) *Um exemplo de entrada de dados e sua distribuição entre $p = 4$ processadores e com o passo 1 do algoritmo já executado. Em cada processador teremos:*

T = *bbbaaaabbabbabbaaabbbbbbaaabbbbbbaaabbbbbbaaaaabbbbbbaaabbbbbbaaabbbbbbaaaaaa bbbbbaaaaaabbbbbbbbaaaabbbbbbbbaaaabbbbbbbbaaaaabbbbbbaaaaabbbbbbaaaabbbbbbbbaaaa bbbbbbbbaaaabbbbbbbbaaaabbbbbbbbaaaaaabbbbbbbbbbbbaaaaaabbbbbbbbbbbbaaaaaaa*

$P = abbabbabba$

0													
	i	1	2	3	4	5	6	7	8	9	10	11	12
	S^Σ	b	a	b	a	b	a	b	a	b	a	b	a
	$S^\#$	3	4	2	1	2	1	2	3	6	3	6	2
	S'	4/3	2/4	1/2	2/1	1/2	2/1	3/2	6/3	3/6	6/3	2/6	

1													
	i	1	2	3	4	5	6	7	8	9	10	11	12
	S^Σ	a	b	a	b	a	b	a	b	a	b	a	b
	$S^\#$	1	6	5	6	3	6	3	6	6	5	6	8
	S'	6/1	5/6	6/5	3/6	6/3	3/6	6/3	6/6	5/6	6/5	8/6	

2													
	i	1	2	3	4	5	6	7	8	9	10	11	12
	S^Σ	a	b	a	b	a	b	a	b	a	b	a	b
	$S^\#$	4	8	4	8	5	5	5	6	4	6	3	5
	S'	8/4	4/8	8/4	5/8	5/5	5/5	6/5	4/6	6/4	3/6	5/3	

3													
	i	1	2	3	4	5	6	7	8	9	10	11	12
	S^Σ	b	a	b	a	b	a	b	a	b	a	b	a
	$S^\#$	3	4	8	4	8	4	8	6	12	6	12	7
	S'	4/3	8/4	4/8	8/4	4/8	8/4	6/8	12/6	6/12	12/6	7/12	

Em todos os processadores teremos:

i	1	2	3	4	5	6	7
P^Σ	a	b	a	b	a	b	a
$P^\#$	1	2	1	2	1	2	1
P'	1/2	2/1	1/2	2/1			

No passo 5, é necessário o envio do primeiro e segundo valores de S^Σ, e do primeiro e segundo valores de $S^\#$, pois, ao se distribuir o texto entre os processadores, símbolos ocupando posições adjacentes em T, podem se tornar o último valor de S^Σ em um processador e o primeiro valor em S^Σ no processador vizinho direito. Os segundos valores devem também ser enviados, pois estas posições não são testadas no procedimento *Fronteira_Esquerda_escala*, como possíveis finais de k-ocorrências. Se esses símbolos são diferentes, não existem grandes problemas como veremos. Porém, se esses símbolos são iguais, deve-se tomar cuidado ao combinar os dados recebidos e os calculados, na obtenção de possíveis k-ocorrências na fronteira. Isto será visto na descrição do procedimento *Combina_escala*. Primeiramente, vamos mostrar como ficam os outros procedimentos utilizados no algoritmo.

No passo 1 do algoritmo *Busca_com_Escala_CGM*, são obtidos os vetores de símbolos e de expoentes, e também os vetores de quocientes, como pode ser visto no exemplo 4(a) e os vetores π e π', no exemplo 4(b), em cada processador.

No passo 3, são calculados os vetores σ e σ' para os vetores P^Σ e P', respectivamente, cuja descrição é análoga ao caso sem escala. O resultado da obtenção destes vetores está no

exemplo 4(b).

Exemplo 4(b) *Resultado da obtenção dos vetores π, π', σ e σ'. Em cada processador temos:*

i	1	2	3	4	5	6	7
π	0	0	1	2	3	4	5
σ	3	4	5	6	7	8	8

i	1	2	3	4
π'	0	0	1	2
σ'	3	4	5	5

No passo 2, o procedimento *Fronteira_Direita_escala* utiliza as mesmas idéias do procedimento *Fronteira_Direita* (página 35) e do algoritmo *Busca_com_Escala* (página 44), nos vetores de símbolos e de quocientes. Algumas pequenas diferenças em relação ao caso sem escala aparecem neste procedimento. No caso sem escala, ao executarmos o algoritmo *KMP* nos dados locais, obtemos as ocorrências, testando até a posição $n - m + 1$ e obtemos as possíveis ocorrências na fronteira a partir da posição $n - m + 2$; no caso escalar, o algoritmo *Busca_com_Escala* testa as posições até $\hat{n} - \hat{m}$ e na obtenção da fronteira direita testamos a partir da posição $\hat{n} - \hat{m} + 1$, isto porque o último símbolo pode ser igual ao primeiro símbolo do processador vizinho direito. Além disso, neste procedimento uma possível ocorrência só é procurada até a posição $\hat{n} - 2$. As duas últimas posições dependem dos dados vindos de outro processador, como veremos no procedimento de combinação dos resultados. Ao final deste passo, teremos uma lista de posições i, $\hat{n} - \hat{m} + 1 \leq i \leq \hat{n} - 2$, com os respectivos valores de k obtidos pela razão entre $S^{\#}[i+1]$ e $P^{\#}[2]$, candidatas a início de uma k-ocorrência. Em cada posição i armazenada nesta lista existe uma ocorrência de $P^{\Sigma}[1 \ldots \hat{n} - i + 1]$ em $S^{\Sigma}[i \ldots \hat{n}]$, uma ocorrência de $P'[1 \ldots \hat{n} - i - 2]$ em $S'[i + 1 \ldots \hat{n} - 2]$ e $S^{\#}[i] \geq k \cdot P^{\#}[1]$. Satisfazendo as propriedades A, B.1 e B.2 da observação 1 para as posições em C_D. No exemplo 4(c), temos a lista C_D construída.

No passo 4, o procedimento *Fronteira_Esquerda_escala* apresenta comportamento simétrico ao *Fronteira_Direita_escala*. A posição $\hat{m}$ é testada por este procedimento e pelo algoritmo *Busca_com_Escala*, pois o primeiro elemento de S^{Σ} pode ser igual ao último símbolo armazenado no processador vizinho esquerdo. Este procedimento devolve uma lista C_E com as posições i, $3 \leq i \leq \hat{m}$, e os correspondentes valores de k obtidos pela razão entre $S^{\#}[i-1]$ e $P^{\#}[\hat{m} - 1]$, que são possíveis finais de k-ocorrências que se iniciam na fronteira direita do processador vizinho esquerdo. Uma posição i em C_E corresponde a uma ocorrência de $P^{\Sigma}[\hat{m} - i + 1 \ldots \hat{m}]$ em $S^{\Sigma}[1 \ldots i]$ (testando, assim todos os símbolos até a posição i) e a uma ocorrência de $P'[\hat{m} - i + 1 \ldots \hat{m} - 3]$ em $S'[2 \ldots i - 2]$; e $T^{\#}[i] \geq k \cdot P^{\#}[\hat{m}]$. No exemplo 4(c) encontram-se a lista C_E e o envio dos dados para os processadores vizinhos esquerdos.

Exemplo 4(c) *Listas R, C_D e C_E calculadas em cada processador e envio para os vizinhos esquerdos.*

No processador 0:
$R = 7, 1$
$C_D = 8, 3 \rightarrow 10, 3$
C_E não é calculado.

No processador 1:
$R = 10, 3$
$C_D = \emptyset$
$C_E = 3, 3$

1 envia para 0:
$C_E = 3, 3$
$T^{\Sigma}[1] = a$, $T^{\#}[1] = 1$
$T^{\Sigma}[2] = b$, $T^{\#}[1] = 6$

No processador 2:
$R = \emptyset$
$C_D = 9, 3$
$C_E = 3, 4 \rightarrow 5, 4$

2 envia para 1:
$C_E = 3, 4 \rightarrow 5, 4$
$T^{\Sigma}[1] = a$, $T^{\#}[1] = 4$
$T^{\Sigma}[2] = b$, $T^{\#}[1] = 8$

No processador 3:
$R = 4, 4$
C_D *não é calculado.*
$C_E = 4, 4 \rightarrow 6, 4$

3 envia para 2:
$C_E = 4, 4 \rightarrow 6, 4$
$T^{\Sigma}[1] = b$, $T^{\#}[1] = 3$
$T^{\Sigma}[2] = a$, $T^{\#}[1] = 4$

Ao iniciar o passo 5, o processador corrente terá recebido: $S^{\Sigma}[1]$ e $S^{\Sigma}[2]$, que consideraremos armazenados como $Simb_1$ e $Simb_2$; $S^{\#}[1]$ e $S^{\#}[2]$, que consideraremos armazenados, respectivamente, como Exp_1 e Exp_2. No procedimento *Combina_escala* temos dois casos:

1. $S^{\Sigma}[\hat{n}] = Simb_1$. Neste caso, definimos $Exp\_ultimo = S^{\#}[\hat{n}] + Exp_1$, armazenando o valor global do expoente do último símbolo.
2. $S^{\Sigma}[\hat{n}] \neq Simb_1$. Neste caso, definimos $Exp\_ultimo = S^{\#}[\hat{n}]$ armazenando o valor global do expoente do último símbolo.

No exemplo 4(d), ilustramos o recebimento dos dados, o cálculo de *Exp_ultimo* e a lista R final.

Exemplo 4(d) *Recebimento dos dados e obtenção da lista R final.*

Processador 0 recebe:
$C_E = 3, 3$
$Simb_1 = a$, $Exp_1 = 1$
$Simb_2 = b$ *e* $Exp_2 = 6$

Processador 1 recebe:
$C_E = 3, 4 \rightarrow 5, 4$
$Simb_1 = a$, $Exp_1 = 4$
$Simb_2 = b$ *e* $Exp_2 = 8$

Processador 2 recebe:
$C_E = 4, 4 \rightarrow 6, 4$
$Simb_1 = b$, $Exp_1 = 3$
$Simb_2 = a$ *e* $Exp_2 = 4$

- *No processador 0.*
  Como $S^{\Sigma}[12] = Simb_1$*:* $Exp\_ultimo = S^{\#}[12] + Exp_1 = 2 + 1 = 3$, $d \leftarrow elemento(C_D)$,

$e \leftarrow elemento(C_E)$, $(n-d)+e = 7 = \hat{m}$, e o valor de k em ambas posições é o mesmo ($k = 3$), o elemento é mantido em C_D. Como o elemento de C_E é removido, C_E fica vazio e os elementos restantes de C_D são removidos. Assim, a lista R final neste processador é $R = 7, 1 \rightarrow 16, 3$.

- *No processador 1.*
  Como $S^{\Sigma}[12] \neq Simb_1$: $Exp\_ultimo = S^{\#}[12] = 8$, $k = 4$. Como C_D é vazio, os elementos de C_E não são percorridos. Como $\hat{m} - 2 = 5$ está armazenada em C_E verificamos que $S^{\Sigma}[11] = P^{\Sigma}[1] = a$, que $S^{\#}[11](= 6) \geq k \cdot P^{\#}[1](= 4 \cdot 1)$, que $S^{\Sigma}[12] = P^{\Sigma}[2] = b$, que $S^{\#}[12](= 8) = k \cdot P^{\#}[2](= 4 \cdot 2 = 8)$, que $Simb_1 = P^{\Sigma}[3] = a$ e que $Exp_1(= 4) = k \cdot P^{\#}[3](= 4 \cdot 1)$. Assim, o par $50, 4$ é incluído na lista R. Portanto, a lista R, neste processador, é $R = 10, 3 \rightarrow 50, 4$.

- *No processador 2.*
  Como $S^{\Sigma}[12] = Simb_1$: $Exp\_ultimo = S^{\#}[12] + Exp_1 = 5 + 3 = 8$, $d \leftarrow elemento(C_D)$, $e \leftarrow elemento(C_E)$, $(n-d)+e = 7 = \hat{m}$, mas o valor de k é diferente, o elemento de C_D é removido, ficando C_D vazio. Nenhuma outra posição é analisada. Logo, o valor final de R neste processador é $R = \emptyset$.

- *No processador 3, não são feitos cálculos adicionais, uma vez que não recebe valores de nenhum outro processador e nas últimas $\hat{m} - 1$ posições não pode haver k-ocorrência. Assim, o resultado final de R é $R = 4, 4$.*

Em ambos os casos, após a determinação de *Exp_ultimo*, podemos verificar quais das posições da fronteira direita, armazenadas em C_D, candidatas a k-ocorrência são certificadas pelos dados recebidos. Cada posição da fronteira direita que for confirmada como k-ocorrência terá a posição correspondente em S_i inserida na lista R.

Em cada processador i, $0 \leq i < p-1$, primeiramente, vamos verificar a certificação quando a k-ocorrência é a posição $\hat{n} - \hat{m} + 1$ para ambos os casos. O expoente do último elemento da k-ocorrência deve ser comparado com *Exp_ultimo*, isto é, deve-se verificar se a desigualdade $Exp\_ultimo \geq k \cdot P^{\#}[\hat{m}]$ ocorre.

A próxima posição a ser verificada é a posição $\hat{n} - \hat{m} + 2$ caso seja candidata a k-ocorrência. Para o caso 1, deve-se verificar se $Exp\_ultimo = k \cdot P^{\#}[\hat{m} - 1]$, se $Simb_2 = P^{\Sigma}[\hat{m}]$ e se $Exp_2 \geq k \cdot P^{\#}[\hat{m}]$. Para o caso 2, verifica-se $Exp\_ultimo = k \cdot P^{\#}[\hat{m} - 1]$, se $Simb_1 = P^{\Sigma}[\hat{m}]$ e se $Exp_1 \geq k \cdot P^{\#}[\hat{m}]$.

Estes dois passos descritos nos parágrafos anteriores são necessários pois, como vimos, as posições armazenadas em C_E não contemplam as duas primeiras posições de S^{Σ}.

Para o caso 2, precisamos ainda trabalhar com a posição $\hat{n} - \hat{m} + 3$ separadamente, quando esta é uma posição de possível k-ocorrência. Aqui, verificamos se $Simb_1 = P^{\Sigma}[\hat{m} - 1]$, se $Exp_1 = k \cdot P^{\#}[\hat{m} - 1]$, se $Simb_2 = P^{\Sigma}[\hat{m}]$ e se $Exp_2 \geq k \cdot P^{\#}[\hat{m}]$.

Nas posições analisadas nos parágrafos anteriores, para ambos os casos, se os resultados das verificações forem verdadeiros, os valores correspondentes em S_i e os respectivos valores de k serão inseridos nas listas R em cada processador.

Após estas verificações em separado, em ambos os casos, partimos da posição seguinte percorrendo C_D e a partir da posição 3 percorremos C_E como se estas listas fossem intercaladas e, como no procedimento *Combina* (página 37), ao invés de intercalá-las, suas distâncias até o final e até o começo de S^Σ, respectivamente, são somados e comparados com o valor de $\hat{m}$ (no caso 1) e $\hat{m}-1$ (no caso 2).

Faltam ainda as posições $\hat{n}-1$ e $\hat{n}$ que não são analisadas pelo procedimento *Fronteira_Direita_escala* e que também serão analisadas separadamente. Vamos, primeiramente, ver como proceder para a posição $\hat{n}-1$. No caso 1, se a posição $\hat{m}-1$ estiver armazenada em C_E, verifica-se se $S^\Sigma[\hat{n}-1] = P^\Sigma[1]$, se $S^\#[\hat{n}-1] \geq k \cdot P^\#[1]$, se $S^\Sigma[\hat{n}] = P^\Sigma[2]$ e se $Exp\_ultimo = k \cdot P^\#[2]$. No caso 2, se a posição $\hat{m}-2$ fizer parte da lista C_E, verificamos se $S^\Sigma[\hat{n}-1] = P^\Sigma[1]$, se $S^\#[\hat{n}-1] \geq k \cdot P^\#[1]$, se $S^\Sigma[\hat{n}] = P^\Sigma[2]$, se $S^\#[\hat{n}] = k \cdot P^\#[2]$, se $Simb_1 = P^\Sigma[3]$ e se $Exp1 = k \cdot P^\#[3]$. Em ambos os casos, se todas as verificações forem satisfeitas, o valor correspondente em S_i e o valor de k serão incluídos em R.

A última posição a ser analisada em separado é $\hat{n}$. No caso 1, se a posição $\hat{m}$ estiver na lista C_E devemos verificar se $Exp\_ultimo \geq k \cdot P_1^\#$ e se $T_{\hat{n}}^\Sigma = P_1^\Sigma$. No caso 2, se a posição $\hat{m}-1$ estiver em C_E, verificamos se $Exp\_ultimo \geq k \cdot P_1^\#$, se $T_{\hat{n}}^\Sigma = P_1^\Sigma$, se $Simb_1 = P_2^\Sigma$ e se $Exp_1 = k \cdot P_2^\#$. Para esta posição, o resultado afirmativo de todas as comparações implica na inclusão, em R, da posição correspondente em S_i e do respectivo valor de k.

Com isto, obtém-se a confirmação, ou não, das posições na fronteira direita como k-ocorrências.

2.5.1 Tempo e Corretude

O tempo de execução local, número de rodadas de comunicação e memória utilizada pelo procedimento *Busca_com_Escala_CGM* são dados pelo teorema a seguir.

Teorema 15 *O algoritmo Busca_com_Escala_CGM leva tempo seqüencial local $O(N/p)$, usando memória $O(N/p)$ e $O(1)$ rodadas de comunicação em que são trocados $O(N/p)$ dados.*

Prova. No passo 1 leva-se tempo $O(\hat{n}+\hat{m})$. Nos passos 2, 3 e 4 leva-se tempo $O(\hat{m})$. No passo 6, o procedimento *Combina_escala* tem comportamento análogo ao procedimento *Combina* (página 37) e leva também tempo $O(\hat{m})$. Nestes passos não há troca de dados e o tempo seqüencial local é $O(\hat{n}+\hat{m})$.

No passo 5, tem-se o envio de dados onde são enviados $O(\hat{m})$ dados em uma rodada de comunicação.

No decorrer do algoritmo temos que armazenar S_i, S^Σ, $S^\#$ e S', gastando memória $O(\hat{n})$. Os vetores P, P^Σ, $P^\#$ e P' consomem memória $O(\hat{m})$. As listas R, C_D e C_E gastam no máximo memória $O(\hat{m})$. Assim, durante o algoritmo gasta-se $O(\hat{n})$ de memória.

Como $\hat{n} = O(N/p)$, $\hat{m} = O(m) = O(N/p)$, o algoritmo tem tempo seqüencial $O(N/p)$, usa uma rodada de comunicação onde são enviados $O(N/p)$ dados, consumindo memória $O(N/p)$.

□

A corretude do algoritmo *Busca_com_Escala_CGM* decorre da corretude dos procedimentos utilizados em cada passo. No passo 1, utilizamos o algoritmo *Busca_com_Escala* (página 44) cuja corretude vem das observações 1 e 2. A corretude dos passos seguintes vem das descrições semiformais dos procedimentos. No passo 3, a construção de σ e σ' é feita pelo procedimento *Constrói_σ* (página 33). Nos passos 2 e 4, a obtenção das listas C_D e C_E utiliza a idéia do algoritmo *Busca_com_Escala* restrito às posições das fronteiras esquerda e direita. No passo 6, existem posições críticas que não são atingidas pelos procedimentos *Fronteira_Direita_escala* e *Fronteira_Esquerda_escala* e que dependem de alguma informação extra além das listas C_E. O comportamento do procedimento nestas posições se encontra na descrição de Combina na seção anterior e sua corretude, quando estas posições são analisadas, decorre diretamente desta descrição. Nas posições não-críticas, o procedimento *Combina_escala* age de maneira análoga ao procedimento *Combina* (página 37) e sua corretude decorre deste.

2.5.2 Resultados da Implementação

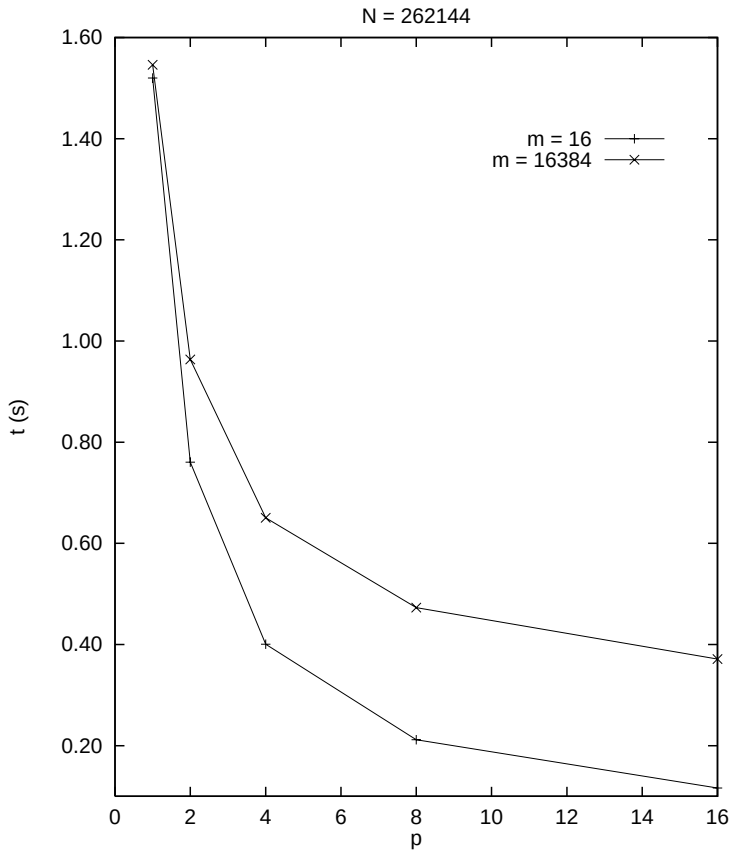

Figura 2.6: Tempo em s para $N = 2^{18} = 262144$, $m = 2^4 = 16$ e $m = 2^{14} = 16384$, utilizando 1, 2, 4, 8 e 16 processadores.

Para este caso, utilizamos uma cadeia de comprimento $N = 2^{18}$ para o texto e cadeias de comprimento $m = 2^4$ e $m = 2^{14}$ para o padrão. Estas cadeias são formadas pelos símbolos a e b alternados, começando por um símbolo a. Neste caso, as ocorrências de escala 1 aparecem em $O(m/2)$ posições, alternadamente. Não existem ocorrências com outras escalas. Este é o caso em que existe o maior número de ocorrências com $\hat{m} > 3$.

Executamos a implementação para estes exemplos utilizando-se 1, 2, 4, 8 e 16 processadores.

Os tempos absolutos para estes dois valores de m podem ser vistos no gráfico da Figura 2.6. Observemos que em ambos os casos a diminuição do tempo é grande, usando-se dois processadores e se torna menos acentuada à medida que o número de processadores aumenta. A quantidade de dados trocados é metade da quantidade trocada no caso sem escala, uma vez que as ocorrências existem em posições alternadas e somente para a escala 1. Além disso, o programa seqüencial é mais complexo e conseqüentemente mais lento. Com isto, o domínio do tempo de comunicação sobre o tempo de computação local é menor.

Podemos observar no gráfico da Figura 2.7 que o *speedup* para $m = 2^4$ é muito bom e para $m = 2^{14}$ este *speedup* é mais discreto, ainda por causa da absorção do tempo de computação local pelo tempo de comunicação.

No apêndice A estão tabulados os valores dos tempos absolutos e dos *speedups*.

2.6 Notas

Neste capítulo, apresentamos algoritmos seqüenciais lineares no tamanho da entrada para os problemas de busca unidimensional de padrões com e sem escala. Para ambos os algoritmos, considerando p processadores, um texto de tamanho N e um padrão de tamanho m, $m \leq N/p$, descrevemos algoritmos CGM que têm tempo de computacao local $O(N/p)$, consumindo memória $O(N/p)$ e utilizam uma rodada de comunicação onde são trocados no máximo $O(m)$ dados.

Uma versão mais simples para ambos os casos seria obtida enviando-se toda a fronteira esquerda para o vizinho esquerdo, antes do início da computação local e estendendo os algoritmos a estas posições. Porém, isto acarretaria uma quantidade fixa e igual a $O(m)$ de dados a serem comunicados. Na nossa versão, a quantidade de dados trocados depende da quantidade de posições de confirmação e é no máximo $O(m)$, sendo, na maioria dos casos, menor que m. Os algoritmos descritos mantêm o tempo teórico de computação local e se beneficiam desta menor quantidade de dados.

Para o caso sem escala, temos uma versão onde apenas o valor da variável q é enviado para o vizinho esquerdo e, após recebê-lo, as posições de ocorrência da fronteira direita são determinadas. Apesar da eficiência desta versão, ela não é extensível para os casos com escala. Para os casos bidimensionais sem escala, esta versão forçaria uma quantidade mínima de dados a serem enviados ($= O(N/p)$), que pode ser maior que o número de posições de confirmação. Assim, por uma questão de uniformidade, mantivemos nossa descrição onde são enviados no máximo $O(m)$ dados.

Os resultados experimentais para o caso sem escala foram relevantes para um padrão de

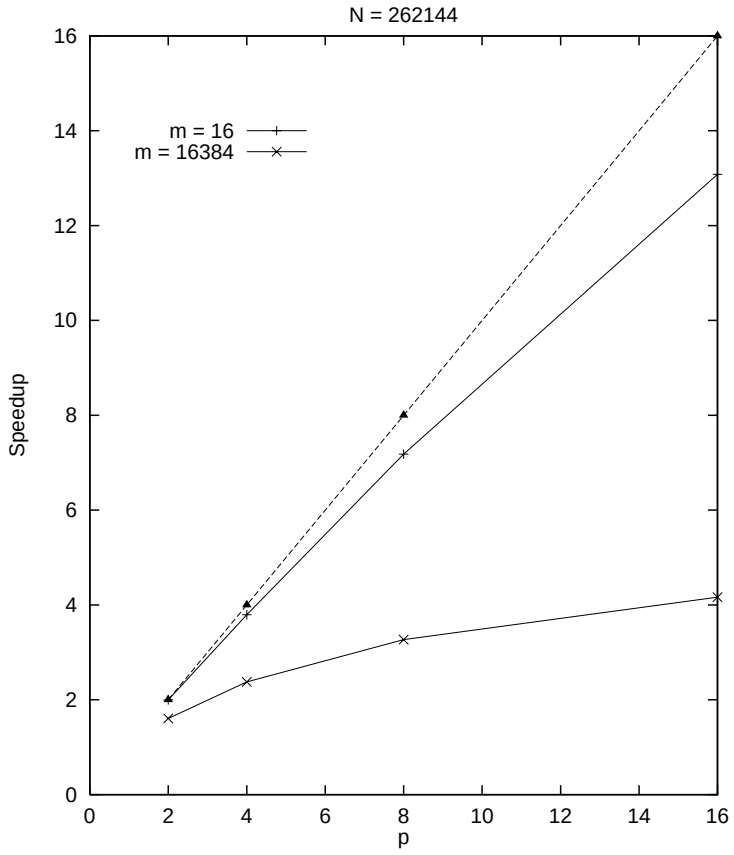

Figura 2.7: *Speedup* para 2, 4, 8 e 16 processadores com $N = 2^{18} = 262144$, $m = 2^4 = 16$ e $m = 2^{14} = 16384$.

comprimento pequeno e mais discretos para padrões de comprimento próximo ao comprimento do texto localmente armazenado. Isto é devido, principalmente, à rapidez do algoritmo seqüencial, de forma que o tempo de computação local acaba sendo bem menor que o tempo de comunicação.

No caso com escala, estas diferenças de comportamento para padrões de diferentes tamanhos foi menor, uma vez que o algoritmo para este caso é mais complexo e a quantidade de dados trocados, no pior caso, é menor e no máximo $O(m/2)$.

Capítulo 3

Busca Bidimensional de Padrões sem Escala

3.1 Introdução

A busca bidimensional de padrões consiste em, dadas uma matriz texto T de ordem N e uma matriz padrão P de ordem m, $m \leq N$, determinar todas as posições de T onde a matriz P ocorre, ou seja, todas as posições de T onde existe uma submatriz igual a P. Por uma questão de simplicidade, consideramos matrizes texto e padrão quadradas. Os algoritmos funcionam igualmente para matrizes retangulares.

Para este problema existem diversos algoritmos seqüenciais, como pode ser visto nas referências [3, 7, 9, 10, 11, 14, 20, 21, 23, 29, 34]. O algoritmo da seção 3.2 leva tempo linear no tamanho da entrada e é descrito por Amir *et al* [8]. Este algoritmo consiste em considerar cada linha do padrão como um símbolo e as cadeias, de comprimento m, começando em cada posição do texto também como símbolos. O problema se reduz, então, a considerar cada coluna como uma cadeia e efetuar-se nela uma busca do padrão, utilizando o algoritmo KMP. Para obter-se a linearidade no tempo de execução, a comparação entre uma linha do padrão e uma cadeia, começando em alguma posição do texto, é feita utilizando-se o resultado da seção 1.9, que garante tempo constante na comparação de duas subcadeias ao utilizar uma árvore de sufixos e consultas LCA nesta.

Diversos algoritmos paralelos no modelo PRAM para este problema estão presentes na literatura [4, 5, 7, 19, 22, 40, 44].

No modelo CGM, o algoritmo proposto na seção 3.3 é o primeiro algoritmo em um modelo de granularidade grossa para este problema. Por causa da limitação no tamanho da memória local dos processadores, alguns cuidados devem ser tomados na distribuição dos dados. Tendo os dados sido distribuídos, o algoritmo seqüencial da seção 3.2 será executado localmente. Existirão ocorrências que terão suas posições em regiões cujos dados não estarão totalmente contidos localmente no processador. Para estes casos utilizaremos uma extensão do conceito de fronteira

do caso unidimensional e mostraremos como lidar com estas regiões. Este algoritmo apresenta tempo local linear e utiliza apenas uma rodada de comunicação. Com a implementação deste algoritmo, obtivemos os resultados experimentais descritos na seção 3.3.5, onde podemos observar um *speedup* muito bom para padrões de ordem pequena em relação à ordem do texto.

Apresentaremos, na seção 3.4, um algoritmo de tempo sublinear descrito também por Amir *et al* [8]. Esta sublinearidade depende da forma como os dados de entrada são apresentados, como veremos em sua descrição. Nele, para cada coluna, determinamos posições onde podem existir ocorrências do padrão e somente estas posições são testadas, utilizando as idéias do algoritmo linear. Este algoritmo serve como base para o algoritmo CGM descrito no capítulo seguinte para o caso com escala.

Para este algoritmo também temos uma versão CGM, descrita na seção 3.5, que leva tempo local linear e utiliza uma rodada de comunicação. A descrição deste algoritmo é similar à do algoritmo CGM da seção 3.2. Em cada processador é executado o algoritmo seqüencial sublinear com os dados locais. No processamento das regiões de fronteira são necessárias algumas adaptações. Os resultados experimentais obtidos com a implementação deste algoritmo podem ser acompanhados na seção 3.5.4 onde também temos um *speedup* significativo para padrões de ordem pequena.

3.2 Um Algoritmo Seqüencial Linear

O algoritmo que apresentaremos foi descrito por Amir *et al* [8] e é uma extensão do algoritmo KMP [41] para o caso bidimensional. Novamente temos duas fases: uma de análise do padrão e uma de análise do texto. Neste algoritmo, o maior problema para se manter a complexidade de tempo linear no tamanho da entrada é a forma como a comparação entre as cadeias será feita. Na descrição do algoritmo, esta comparação entre cadeias será feita em tempo constante com as estruturas de dados obtidas em uma etapa de pré-processamento.

Vamos considerar como entrada uma matriz texto T e uma matriz padrão P. Para facilitar a exposição consideramos que estas matrizes são quadradas e de ordem N e m, respectivamente, com $|T| = N^2$ e $|P| = m^2$. O algoritmo funciona de maneira análoga para matrizes retangulares.

3.2.1 Pré-Processamento da Entrada

O primeiro passo deste algoritmo compreende o pré-processamento da entrada. Nesta etapa, construímos uma estrutura de dados para que comparações entre cadeias possam ser feitas em tempo constante. Inicialmente, construímos uma cadeia C formada pela concatenação de todas as linhas de T e anexada a ela a concatenação de todas as linhas de P. A partir desta cadeia, construímos a árvore de sufixos ST de C (seção 1.6). A esta árvore ST, aplicamos um algoritmo para o problema do LCA (seção 1.7).

As consultas, que aparecem no algoritmo, compreendem a comparação de duas subcadeias de C de comprimento m, ou seja, $c_i, c_{i+1}, \ldots, c_{i+m-1}$ e $c_j, c_{j+1}, \ldots, c_{j+m-1}$. A resposta a esta

comparação se reduz a determinar se o prefixo comum mais longo dos sufixos $c_i, c_{i+1}, \ldots$ e $c_j, c_{j+1}, \ldots$ de C tem comprimento maior que $m-1$. Esta consulta é respondida através de uma consulta LCA à árvore ST, e pode ser feita em tempo constante.

3.2.2 Análise do Padrão

Para esta etapa, temos a estrutura de dados construída no passo anterior. Aplicamos, então, a etapa de análise do padrão, como na seção 2.2.1, para a cadeia $\bar{P} = \bar{P}[1], \bar{P}[2], \ldots, \bar{P}[m]$, considerando cada linha de P como um símbolo de $\bar{P}$, isto é, $\bar{P}[i] = P[i; 1, \ldots, m]$. A comparação $\bar{P}[i] = \bar{P}[j]$ de dois símbolos de $\bar{P}$ se reduz à comparação das linhas i e j de P, que é feita em tempo $O(1)$, utilizando-se a estrutura de dados da etapa de pré-processamento.

Assim, esta análise de padrão compreende a obtenção de um vetor π de m posições. O procedimento que calcula este vetor é similar ao procedimento da seção 2.2.1, a diferença se encontra na comparação dos símbolos como descrito acima. Ao compararmos dois símbolos de $\bar{P}$ (ou, equivalentemente, duas linhas de P), utilizaremos o procedimento *Igual*, cujos argumentos são duas cadeias de comprimento m, que retorna 1 se elas forem iguais e 0, caso contrário. Este procedimento é executado em tempo constante, pois o que ele faz é apenas uma consulta LCA na árvore ST de C.

PROCEDIMENTO: Constrói2_π.
Entrada: a matriz padrão P.
Saída: o vetor π de m posições.
{ Consideramos que já foram obtidas a árvore de sufixos ST e a estrutura de dados para as consultas LCA. }

1. $m \leftarrow Ordem(P)$
2. $\pi[1] \leftarrow 0$
3. $k \leftarrow 0$
4. **para** $q \leftarrow 2$ **até** m **faça**
5. **enquanto** $k > 0$ **e** $\neg Igual(P[q; 1, \ldots, m], P[k+1; 1, \ldots, m])$ **faça**
6. $k \leftarrow \pi[k]$
7. **se** $Igual(P[q; 1, \ldots, m], P[k+1; 1, \ldots, m])$
8. $k \leftarrow k + 1$
9. $\pi[q] \leftarrow k$
10. **devolva** π

fim procedimento

Pode-se ver facilmente que o procedimento leva tempo $O(m)$, pois o laço **para** da linha 4 é repetido $m-1$ vezes e o corpo deste laço tem tempo amortizado de pior caso $O(1)$, como no procedimento *Constrói_π* (página 25). A corretude do procedimento decorre da corretude do procedimento *Constrói_π* (página 25) e das observações da seção 1.9, pois cada linha é vista como um símbolo e a comparação entre eles utiliza o procedimento *Igual*.

3.2.3 Análise do Texto

A análise do texto também é análoga ao algoritmo *KMP* (página 25). O único detalhe é que a comparação entre o início de uma subcadeia em T e uma linha em P é feita, usando-se o procedimento *Igual*, como podemos ver na descrição a seguir. Novamente, vamos armazenar a saída em uma lista ordenada R de pares $[i, j]$. Nesta lista ficarão as posições lexicograficamente ordenadas por i e depois por j.

ALGORITMO: KMP_2
Entrada: a matriz texto T e a matriz padrão P.
Saída: uma lista R com as posições de T onde P ocorre.

1. $N \leftarrow Ordem(T)$
2. $m \leftarrow Ordem(P)$
3. $C \leftarrow Concatena(T, P)$
4. $T \leftarrow Constrói\_ST(C)$
5. $LCA(T)$
6. $\pi \leftarrow Constrói2\_\pi(P)$
7. $q \leftarrow 0$
8. **para** $j \leftarrow 1$ **até** $N - m + 1$ **faça**
9. **enquanto** $i \leq N$ **faça**
10. **enquanto** $q > 0$ **e** $\neg Igual(P[q+1; 1, \ldots, m], T[i; j, \ldots, j+m-1])$ **faça**
11. $q \leftarrow \pi[q]$
12. **se** $Igual(P[q+1; 1, \ldots, m], T[i; j, \ldots, j+m-1])$
13. $q \leftarrow q + 1$
14. **se** $q = m$
15. $Insere((i - m + 1, j), R)$
16. $q \leftarrow \pi[q]$

fim algoritmo

Este algoritmo leva tempo $O(N^2 + m^2)$, pois os passos 3, 4 e 5 levam tempo $O(N^2 + m^2)$. A linha 6 leva tempo $O(m)$. Os laços **para** da linha 8 e **enquanto** da linha 9 são executados $O(N)$ vezes cada um e o corpo deste último laço leva tempo $O(1)$. Logo, o laço **para** da linha 8 tem complexidade de pior caso $O(N^2)$. Assim, o algoritmo roda em tempo linear no tamanho da entrada. A corretude deste algoritmo segue diretamente da corretude dos procedimentos utilizados e da corretude do algoritmo *KMP* (página 25), do qual este é derivado.

3.3 O Algoritmo CGM Baseado no Algoritmo Seqüencial Linear

Considere uma matriz texto T de ordem N e uma matriz padrão P de ordem m, com $|T| = N^2$ e $|P| = m^2$. Vamos apresentar um algoritmo CGM para a busca bidimensional de

padrões baseado no algoritmo seqüencial da seção anterior. Antes de apresentar o algoritmo, precisamos determinar como os dados serão distribuídos entre os processadores.

3.3.1 Distribuição dos Dados

Consideraremos que o padrão estará armazenado em cada processador. Temos que $|P| = O(\frac{|T|}{p})$, e portanto

$$m^2 \leq \frac{N^2}{p} \Leftrightarrow m \leq \frac{N}{\sqrt{p}} \Leftrightarrow p \leq \frac{N^2}{m^2}.$$

A forma de armazenamento do texto nos processadores não é tão óbvia e imediata como no caso unidimensional. Isto porque a quantidade de informações trocadas em cada etapa de comunicação é limitada pelo modelo e em nosso algoritmo a quantidade de dados é $O(\frac{|T|}{p}) = O(\frac{N^2}{p})$.

Poderíamos dividir o texto em faixas verticais de largura N/p ou horizontais de altura também N/p e atribuir cada faixa a um processador. Entretanto, precisaríamos enviar no máximo $(m-1)N$ dados entre dois processadores, na única rodada de comunicação do algoritmo, com $m = O(N/\sqrt{p})$, e teríamos $O(N^2/\sqrt{p})$ dados sendo enviados, o que viola a quantidade máxima de dados enviados imposta pelo modelo.

A solução é cada processador receber uma submatriz de T de dimensões $\frac{N}{\sqrt{p}} \times \frac{N}{\sqrt{p}}$. Para facilitar a exposição e as demonstrações vamos considerar, sem perda de generalidade, que p é um quadrado perfeito e $\sqrt{p}$ é um divisor de N. Vamos definir $n = \frac{N}{\sqrt{p}}$.

Com esta distribuição dos dados, teremos o seguinte mapeamento que associa cada submatriz aos processadores: a submatriz $S[(i-1)n+1, \ldots, in; (j-1)n+1, \ldots, jn]$ estará armazenada no processador de índice k, ou simplesmente processador k, com $k = (i-1)\sqrt{p} + j - 1$, onde $1 \leq i, j \leq \sqrt{p}$ e $0 \leq k \leq p-1$, e será denominada submatriz $S_{i,j}$.

O conceito de vizinhança entre processadores, vista no caso unidimensional na seção 2.3, é um pouco mais complexo neste caso. A Figura 3.1 ilustra a vizinhança entre $p = 25$ processadores. As submatrizes armazenadas nos processadores k com $\sqrt{p} < k < p - \sqrt{p}$, $k \bmod \sqrt{p} > 0$ e $(k+1) \bmod \sqrt{p} > 0$, têm submatrizes adjacentes:

- à direita, que estará no processador $k+1$, denominado *vizinho Leste (E)*;
- à esquerda, armazenada em $k-1$ e denominado *vizinho Oeste (O)*;
- abaixo, que estará no processador $k + \sqrt{p}$ denominado *vizinho Sul (S)*; e
- acima, armazenado no processador $k - \sqrt{p}$, denominado *vizinho Norte (N)*.

Estas submatrizes ainda se relacionam com outras não-adjacentes mas que estão nas diagonais:

- a submatriz armazenada no processador $k + \sqrt{p} + 1$, denominado *vizinho Sudeste (SE)*;

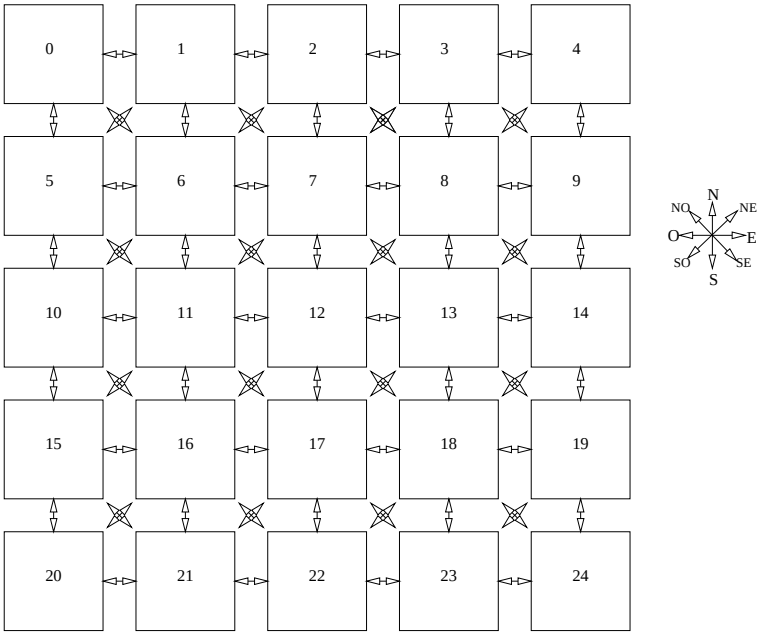

Figura 3.1: Representação das vizinhanças entre processadores para os dados distribuídos em $p = 25$ processadores.

- a armazenada no processador $k + \sqrt{p} - 1$, denominado *vizinho Sudoeste (SO)*;
- a submatriz armazenada no processador $k - \sqrt{p} - 1$, chamado *vizinho Noroeste (NO)*; e
- a armazenada em $k - \sqrt{p} + 1$, denominado *vizinho Nordeste (NE)*.

Utilizamos os pontos cardeais, como ilustrado na Figura 3.2 para evitar o uso de termos como "diagonal abaixo esquerda" e usaremos suas siglas para identificá-los.

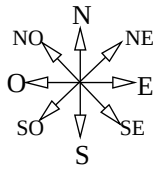

Figura 3.2: Representação dos pontos cardeais.(Rosa-dos-ventos).

As submatrizes não cobertas pelas restrições do índice do processador vistas apresentam menos vizinhos, a saber, as submatrizes armazenadas nos processadores k:

- com $k \bmod \sqrt{p} = 0$, para $0 < k < p - \sqrt{p} - 1$, apresentam vizinhos N, NE, E, SE e S;
- com $0 < k < \sqrt{p} - 1$, têm vizinhos E, SE, S, SO e O;
- com $(k + 1) \bmod \sqrt{p} = 0$, para $\sqrt{p} < k < p - 1$, têm vizinhos S, SO, O, NO e N;

- com $p - \sqrt{p} < k < \sqrt{p} - 1$, possuem vizinhos O, NO, N, NE e E;
- com $k = 0$, tem vizinhos E, SE e S;
- com $k = \sqrt{p} - 1$ tem vizinhos S, SO e O;
- com $k = p - 1$ tem vizinhos O, NO e N; e
- com $p - \sqrt{p}$ tem vizinhos N, NE e E.

Este conceito de vizinhança pode ser estendido às submatrizes e poderemos dizer que uma submatriz $S_{i,j}$ armazenada em um processador k, tem, por exemplo, um vizinho Sudeste $S_{i',j'}$ se este estiver armazenado em $k + \sqrt{p} + 1$ e teremos $i' = i + 1$ e $j' = j + \sqrt{p}$.

Na Figura 3.1 os processadores: 6, 7, 8, 11, 12, 13, 16, 17 e 18 têm vizinhos em todas as direções; 0 tem vizinhos S, E e SE; 4 tem vizinhos S, O e SO; 20 tem vizinhos N, E e NE; 24 tem vizinhos N, O e NO; 1, 2 e 3 têm vizinhos E, SE, S, SO e O; e, 21, 22 e 23 têm vizinhos O, NO, N, NE e E.

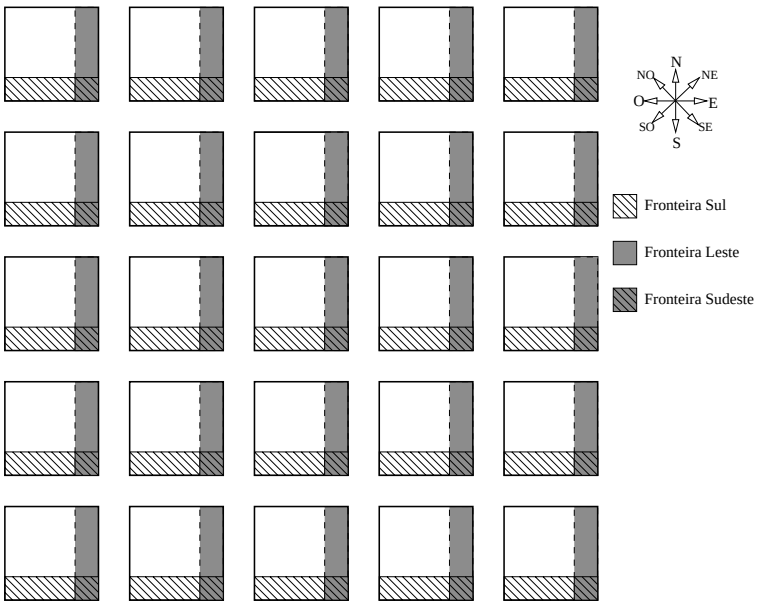

Figura 3.3: Representação das fronteiras Sul(S), Leste(E) e Sudeste(SE).

Ao buscarmos as ocorrências do padrão P na matriz texto T, devemos procurá-las no texto como um todo. Assim, o padrão pode ocorrer inteiramente na submatriz armazenada localmente ou pode ocorrer em uma posição e espalhar-se pelas submatrizes dos processadores vizinhos. Neste último, teremos uma ocorrência do padrão em uma região de fronteira. As *fronteiras* serão submatrizes das matrizes armazenadas localmente e usaremos os pontos cardeais para designá-las. A existência de fronteiras em uma submatriz depende da existência de um processador vizinho correspondente nesta direção. Por exemplo, se uma submatriz tem um vizinho na direção NO, ela terá uma fronteira NO. Assim, as *fronteiras Norte e Sul* são faixas horizontais

$[1,\ldots,m-1;1,\ldots,n]$ e $[n-m+2,\ldots,n;1,\ldots,n]$, respectivamente; as *fronteiras Leste e Oeste* são faixas verticais $[1,\ldots,n;n-m+2,\ldots,n]$ e $[1,\ldots,n;1,\ldots,m-1]$, respectivamente; as *fronteiras Nordeste, Sudeste, Sudoeste e Noroeste* são matrizes quadradas $[n-m+2,\ldots,n;1,\ldots m-1]$, $[n-m+2,\ldots n;n-m+2,\ldots,n]$, $[1,\ldots,m-1;n-m+2,\ldots,n]$ e $[1,\ldots,m-1;1,\ldots m-1]$, respectivamente. As Figuras 3.3 e 3.4 ilustram estas fronteiras.

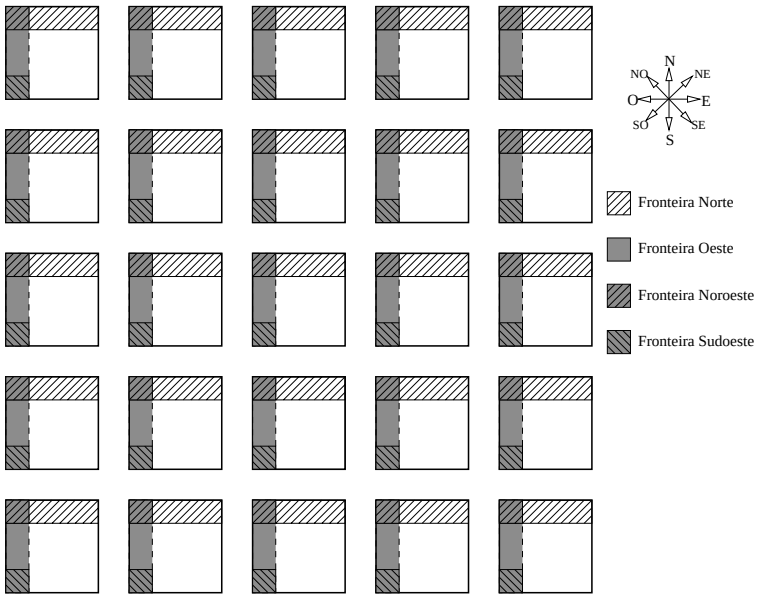

Figura 3.4: Representação das fronteiras Norte(N), Oeste(O) e Noroeste(NO).

3.3.2 Análise do Padrão

Neste modelo, a etapa de análise do padrão utiliza as idéias do algoritmo seqüencial, onde se calcula um vetor π. Por causa da fronteira Leste, as informações contidas neste vetor π não são suficientes. Nestas posições, não são comparadas todas as colunas de P, pois as colunas de j analisadas estão no intervalo $n-m+2 \leq j \leq n$ e $n-j < m$. A determinação de possíveis ocorrências é feita, procurando-se ocorrências da submatriz $P[1,\ldots,m;1,\ldots,j-n+m]$ de P em $S[1,\ldots,n;j,\ldots,n]$ e para isto precisamos de um vetor π para esta submatriz. Nesta etapa, vamos construir uma matriz $\bar{\pi}$, de ordem m, cuja coluna j representa o vetor π para a submatriz. Quando $j=1$, temos o vetor π do caso seqüencial. A seguir, temos o procedimento que obtém $\bar{\pi}$.

PROCEDIMENTO: Constrói_$\bar{\pi}$
Entrada: a matriz padrão P.
Saída: a matriz $\bar{\pi}$ de ordem m.
{ Consideramos que já foram obtidas a árvore de sufixos ST e a estrutura de dados para as consultas LCA. }

1. $m \leftarrow Ordem(P)$
2. **para** $j \leftarrow 1$ **até** m **faça**
3. $\quad \bar{\pi}[1][j] \leftarrow 0$
4. $\quad k \leftarrow 0$
5. $\quad$ **para** $q \leftarrow 2$ **até** m **faça**
6. $\quad\quad$ **enquanto** $k > 0$ **e** $\neg Igual(P[q; 1, \ldots, m-j+1], P[k+1; 1, \ldots, m-j+1])$ **faça**
7. $\quad\quad\quad k \leftarrow \bar{\pi}[k][j]$
8. $\quad\quad$ **se** $Igual(P[q; 1, \ldots, m-j+1], P[k+1; 1, \ldots, m-j+1])$
9. $\quad\quad\quad k \leftarrow k+1$
10. $\quad\quad \bar{\pi}[q][j] \leftarrow k$
11. **devolva** $\bar{\pi}$

fim procedimento

O procedimento leva tempo de pior caso $O(m^2)$, pois temos dois laços aninhados que são executados $O(m)$ vezes cada um, com o corpo do laço interno tendo tempo amortizado de pior caso $O(1)$. A corretude do procedimento decorre diretamente da corretude do procedimento *Contrói2_π* (página 58).

Além da matriz $\bar{\pi}$, temos que calcular uma matriz $\bar{\sigma}$ por causa da fronteira Oeste. Neste caso, estaremos procurando uma ocorrência de $P[1, \ldots, m; m-j+1, \ldots, m]$ para $1 \leq j < m$ em $S[1, \ldots, n; 1, \ldots, j]$ e precisamos de um vetor σ para cada j, que é a coluna j de $\bar{\sigma}$. A descrição do algoritmo é análoga a menos da forma de percorrer as linhas de P, de baixo para cima, e apresenta a mesma complexidade de tempo e a mesma justificativa para a corretude.

São necessárias, ainda, duas outras matrizes obtidas a partir da matriz padrão. A primeira é a matriz $\bar{\pi}'$, que é obtida comparando-se subcadeias das linhas de P entre si, no sentido do crescimento dos índices de linhas, de forma que a coluna j de $\bar{\pi}'$ é o vetor π obtido da matriz $P[1, \ldots, m; j, \ldots, m]$. A segunda matriz é denominada $\bar{\sigma}'$ obtida de forma análoga, considerando-se os índices de linha decrescentes e tal que a coluna j de $\bar{\sigma}'$ armazena o vetor σ da matriz $P[1, \ldots, m; 1, \ldots m-j+1]$. A obtenção destas matrizes é similar à obtenção das anteriores tendo mesma complexidade de tempo e demonstração de corretude análoga.

3.3.3 Análise do Texto

A seguir, descrevemos o algoritmo para o caso bidimensional, cuja seqüência de passos é praticamente a mesma do algoritmo do caso unidimensional. As diferenças residem nas chamadas de procedimentos, que lidam com os dados locais e com as regiões de fronteira no âmbito bidimensional.

Primeiramente, aplica-se o algoritmo de busca bidimensional de padrões aos dados localmente armazenados, obtendo-se uma lista local R com as posições de ocorrência. No passo seguinte, obtemos as *posições candidatas* à ocorrência - aquelas que serão ou não posições (globais) de ocorrência, dependendo dos dados dos processadores vizinhos. Estas posições candidatas são determinadas nas fronteiras Sul, Sudeste e Leste. Em seguida, obtemos as posições que irão servir para confirmar, nos vizinhos, se uma posição candidata é, ou não, uma posição de ocorrência.

Estas posições serão denominadas *posições de confirmação*. As posições de confirmação serão obtidas nas fronteieras Sudoeste, Oeste, Noroeste e Norte, e serão enviadas para os respectivos vizinhos. Com as posições de confirmação devidamente recebidas, utilizamos o procedimento *Combina_Bi*, que determina as posições de ocorrência nas regiões de fronteira, armazenando-as também nas listas R locais.

ALGORITMO: Busca_sem_Escala_Bi_CGM
Entrada: uma matriz texto T, uma matriz padrão P.
Saída: listas R nos processadores com as posições da submatriz local S onde P ocorre.
{ Cada processador de índice $(i-1)\sqrt{p}+j-1$ recebe uma submatriz $S_{i,j}$ de ordem $n=\frac{N}{\sqrt{p}}$ da matriz T, com $1 \leq i,j \leq \sqrt{p}$ e a matriz padrão P. }

1. Cada processador k executa o algoritmo *KMP_2* com os dados locais, obtendo uma lista R.
2. Cada processador k, $0 \leq k < p-1$, executa:
   - 2.1 o procedimento *Fronteira_SSEE*, se $(k+1) \bmod \sqrt{p} > 0$ e $k < p-\sqrt{p}$, obtendo a lista C_{SSEE}.
   - 2.2 o procedimento *Fronteira_S*, se $(k+1) \bmod \sqrt{p} = 0$, obtendo a lista C_S.
   - 2.3 o procedimento *Fronteira_E*, se $k \geq p-\sqrt{p}$, obtendo a lista C_E.
3. Cada processador k, $0 < k \leq p-1$, executa o procedimento *Constroi_σ*.
4. Cada processador k, $0 < k \leq p-1$, executa:
   - 4.1 o procedimento *Fronteira_ONON*, se $k \bmod \sqrt{p} > 0$ e $k > \sqrt{p}$, obtendo as listas C_O, C_{SO}, C_N, C_{NE} e C_{NO}.
   - 4.2 o procedimento *Fronteira_N*, se $k \bmod \sqrt{p} = 0$, obtendo as lista C_N e C_{NE}.
   - 4.3 o procedimento *Fronteira_O*, se $k < \sqrt{p}$, obtendo as listas C_O e C_{SO}.
5. Cada processador k, $0 < k \leq p-1$, envia:
   - 5.1 as listas: C_O e C_{SO} para o processador $k-1$, que recebe em C_O e C_{SO}, respectivamente; C_N e C_{NE} para o processador $k-\sqrt{p}$, que recebe em C_N e C_{NE}, respectivamente; e C_{NO} para o processador $k-\sqrt{p}-1$, que recebe em C_{NO}, onde $k \bmod \sqrt{p} > 0$ e $k > \sqrt{p}$.
   - 5.2 as listas C_N e C_{NE} para o processador $k-\sqrt{p}$, que recebe em C_N e C_{NE}, respectivamente, onde $k \bmod \sqrt{p} = 0$.
   - 5.3 as listas C_O e C_{SO} para o processador $k-1$, que recebe em C_O e C_{SO}, respectivamente, onde $k < \sqrt{p}$.
6. Cada processador k, $0 \leq k < p-1$, executa o procedimento *Combina_Bi*.

fim algoritmo

Os procedimentos do passo 2 constroem as listas de posições candidatas a ocorrência de P em T, que dependem dos dados dos processadores vizinhos para serem confirmadas, ou não, como ocorrências. Por sua vez, os procedimentos do passo 4 constroem as listas de posições de confirmação que servirão para certificar em algum processador vizinho se uma posição candidata é de fato uma posição de ocorrência. No passo 5, as listas com as posições de confirmação são enviadas para os respectivos vizinhos. As posições de confirmação recebidas serão armazenadas em listas que terão o mesmo nome das listas de confirmação localmente calculadas e já enviadas, uma vez que as listas de posições de confirmação não têm mais utilidade após o seu envio. No passo 6, as posições de confirmação são combinadas com as posições candidatas para certificar quais destas últimas realmente são ocorrências do padrão. A seguir, descreveremos os procedimentos *Fronteira_S* e *Fronteira_E*.

PROCEDIMENTO: Fronteira_S
Entrada: uma submatriz de texto S e a matriz padrão P.
Saída: a lista C_S com as posições em que P pode ocorrer em S na fronteira Sul.

1. $C_S \leftarrow \emptyset$
2. $n \leftarrow Ordem(S)$
3. $m \leftarrow Ordem(P)$
4. **para** $j \leftarrow 1$ **até** $n - m + 1$ **faça**
5. $\quad q \leftarrow 0$
6. $\quad$ **para** $i \leftarrow n - m + 2$ **até** n **faça**
7. $\quad\quad$ **enquanto** $q > 0$ **e** $\neg Igual(P[q+1; 1, \ldots, m], S[i; j, \ldots, j+m-1])$ **faça**
8. $\quad\quad\quad q \leftarrow \bar{\pi}[q][1]$
9. $\quad\quad$ **se** $Igual(P[q+1; 1, \ldots, m], S[i; j, \ldots, j+m-1])$
10. $\quad\quad\quad q \leftarrow q + 1$
11. $\quad$ **enquanto** $q > 0$ **faça**
12. $\quad\quad Insere((n - q + 1, j), C_S)$
13. $\quad\quad q \leftarrow \bar{\pi}[q][1]$
14. **devolva** C_S

fim procedimento

No procedimento *Fronteira_S*, percorremos as $m - 1$ últimas linhas e as $n - m + 1$ primeiras colunas verificando se existem candidatos a ocorrência de P em S. Em caso afirmativo, estas posições são inseridas na lista C_S. As posições correspondentes à fronteira Sudeste não serão verificadas neste procedimento.

PROCEDIMENTO: Fronteira_E
Entrada: uma submatriz de texto S e a matriz padrão P.
Saída: a lista C_E com as posições em que P pode ocorrer em S na fronteira Leste.

1. $C_E \leftarrow \emptyset$
2. $n \leftarrow Ordem(S)$
3. $m \leftarrow Ordem(P)$

4. **para** $j \leftarrow n - m + 2$ **até** n **faça**
5. $q \leftarrow 0$
6. **para** $i \leftarrow 1$ **até** n **faça**
7. **enquanto** $q > 0$ **e** $\neg Igual(P[q+1; 1, \ldots, n-j+1], S[i; j, \ldots, n])$ **faça**
8. $q \leftarrow \bar{\pi}[q][j - n + m]$
9. **se** $Igual(P[q+1; 1, \ldots, n-j+1], S[i; j, \ldots, n])$
10. $q \leftarrow q + 1$
11. **se** $q = m$
12. $Insere((i - m + 1, j), C_E)$
13. $q \leftarrow \bar{\pi}[q][j - n + m]$
14. **devolva** C_E

fim procedimento

Este procedimento, por sua vez, verifica a existência de candidatos a ocorrência nas linhas entre 1 e $n - m + 1$, e nas últimas $m - 1$ colunas. As posições, que forem candidatas, serão inseridas na lista C_E. Novamente, as posições correspondentes à fronteira Sudeste não serão analisadas.

O procedimento *Fronteira_SSEE* verifica a ocorrência de candidatos nas fronteiras Sul, Sudeste e Leste. Dessa forma, a descrição deste procedimento é formada pelas descrições de *Fronteira_S* e *Fronteira_E*, sendo que as posições candidatas são inseridas na lista C_{SSEE}, e após o correspondente em *Fronteira_E* ao trecho entre as linhas 6 e 13, dentro do laço **para** da linha 4, é inserido o laço descrito a seguir.

$\vdots$
enquanto $q > 0$ **faça**
$Insere((n - q + 1, j), C_{SSEE})$
$q = \bar{\pi}[q][j - n + m]$
$\vdots$

Neste laço, as posições correspondentes à fronteira Sudeste são percorridas e as candidatas à ocorrência são inseridas na lista C_{SSEE}. Nos procedimentos *Fronteira_S* e *Fronteira_E*, as posições da fronteira Sudeste não são percorridas pois estes procedimentos rodam em processadores para os quais os vizinhos Nordeste não estão definidos

Nos procedimentos anteriores utilizamos a matriz $\bar{\pi}$ na busca dos candidatos. A determinação das posições que servirão para confirmação de ocorrências é feita pelos procedimentos *Fronteira_N*, *Fronteira_O* e *Fronteira_ONON*. Estes procedimentos precisam das matrizes $\bar{\sigma}$, $\bar{\pi}'$ e $\bar{\sigma}'$, construídas no passo 3.

As posições de confirmação obtidas nestes procedimentos estão na diagonal do padrão em relação a uma possível posição candidata, quando esta estiver nas fronteiras Sul ou Leste. Ou seja, a posição de confirmação deve estar $m-1$ linhas abaixo e $m-1$ colunas à direita da posição candidata, em relação à matriz texto original para que esta seja confirmada como ocorrência. A

Figura 3.5 mostra estas relações entre as posições candidatas e de confirmação nestas fronteiras. Para as candidatas nestas fronteiras apenas uma posição de confirmação basta para certificá-las.

Quando a posição candidata estiver na fronteira Sudeste, serão necessárias 3 posições de confirmação: uma que venha do vizinho Sul, uma do vizinho Sudeste e uma do vizinho Leste. A posição que vem do vizinho Sul estará na mesma coluna da posição candidata e $m-1$ linhas abaixo. A posição de confirmação que vem do Leste estará na mesma linha da candidata e $m-1$ colunas à direita desta. E a que vem de Sudeste estará na diagonal do padrão em relação à posição candidata. Assim, para que esta posição candidata seja efetivada como ocorrência, deverá ter estas 3 posições de confirmação em relação a T. Na Figura 3.5 temos uma visualização do que ocorre neste caso.

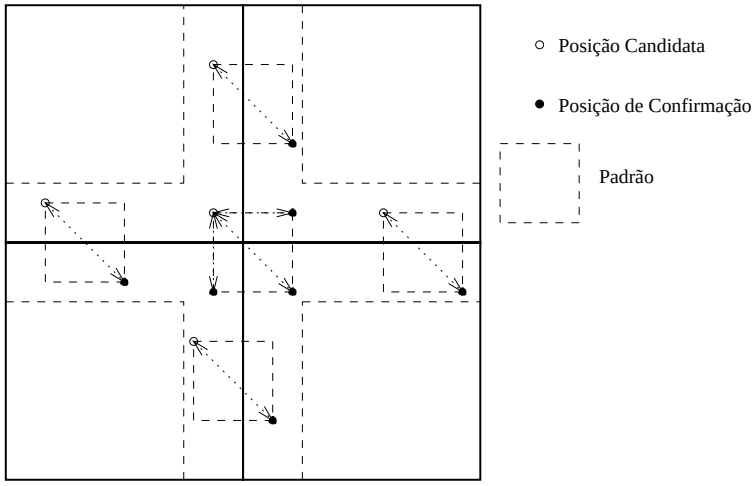

Figura 3.5: Relação entre posições candidatas e de confirmação.

A seguir encontram-se as descrições dos procedimentos *Fronteira_N* e *Fronteira_O*.

PROCEDIMENTO: Fronteira_N
Entrada: uma submatriz de texto S e a matriz padrão P.
Saída: as listas C_N e C_{NE} com as posições em que P pode ocorrer em S nas fronteiras Norte e Nordeste.

1. $C_N \leftarrow \emptyset$
2. $n \leftarrow Ordem(S)$
3. $m \leftarrow Ordem(P)$
4. **para** $j \leftarrow n$ **até** m **passo** -1 **faça**
5. $\quad q \leftarrow m+1$
6. $\quad$ **para** $i \leftarrow m-1$ **até** 1 **passo** -1 **faça**
7. $\quad\quad$ **enquanto** $q < m+1$ **e** $\neg Igual(P[q-1;1,\ldots,m], S[i;j-m+1,\ldots,j])$ **faça**
8. $\quad\quad\quad q \leftarrow \bar{\sigma}[q][1]$
9. $\quad\quad$ **se** $Igual(P[q-1;1,\ldots,m], S[i;j-m+1,\ldots,j])$
10. $\quad\quad\quad q \leftarrow q-1$

11. **enquanto** $q < m+1$ **faça**
12. $Insere((m-q+1, j), C_N)$
13. $q \leftarrow \bar{\sigma}[q][1]$
14. **para** $j \leftarrow n-m+2$ **até** n **faça**
15. $q \leftarrow m+1$
16. **para** $i \leftarrow m-1$ **até** 1 **passo** -1 **faça**
17. **enquanto** $q < m+1$ **e** $\neg Igual(P[q-1; 1, \ldots, n-j+1], S[i; j, \ldots, n])$ **faça**
18. $q \leftarrow \bar{\sigma}'[q][j-n+m]$
19. **se** $Igual(P[q-1; 1, \ldots, n-j+1], S[i; j, \ldots, n])$
20. $q \leftarrow q-1$
21. **enquanto** $q < m+1$ **faça**
22. $Insere((m-q+1, j), C_{NE})$
23. $q \leftarrow \bar{\sigma}'[q][j-n+m]$
24. **devolva** C_N e C_{NE}

fim procedimento

No trecho entre as linhas 4 e 13 do procedimento *Fronteira_N* são computadas as posições de confirmação na submatriz delimitada pelas últimas $n-m+1$ colunas e as $m-1$ primeiras linhas, ou seja, as posições de confirmação na fronteira Norte. Estas posições são percorridas da direita para a esquerda e de baixo para cima, utilizando-se a primeira coluna de $\bar{\sigma}$. As posições de confirmação são armazenadas em C_N.

Nas linhas 14 a 23, percorremos as posições da fronteira Nordeste obtendo as posições que irão certificar, ou não, as posições candidatas na fronteira Sudeste do vizinho Norte. Para se determinar estas posições utilizamos a matriz $\bar{\sigma}'$ e percorremos esta fronteira da esquerda para a direita e de baixo para cima. Estas posições de confirmação são armazenadas na lista C_{NE}.

A saída deste procedimento é formada pelas posições de confirmação para as candidatas das fronteiras Sul e Sudeste do vizinho Norte.

PROCEDIMENTO: Fronteira_O
Entrada: uma submatriz de texto S e a matriz padrão P.
Saída: as listas C_O e C_{SO} com as posições em que P pode ocorrer em S nas fronteiras Oeste e Sudoeste.

1. $C_O \leftarrow \emptyset$
2. $n \leftarrow Ordem(S)$
3. $m \leftarrow Ordem(P)$
4. **para** $j \leftarrow m-1$ **até** 1 **passo** -1 **faça**
5. $q \leftarrow m+1$
6. **para** $i \leftarrow n$ **até** 1 **passo** -1 **faça**
7. **enquanto** $q < m+1$ **e** $\neg Igual(P[q-1; m-j+1, \ldots, m], S[i; 1, \ldots, j])$ **faça**
8. $q \leftarrow \bar{\sigma}[q][m-j+1]$
9. **se** $Igual(P[q-1; m-j+1, \ldots, m], S[i; 1, \ldots, j])$
10. $q \leftarrow q-1$

11. **se** $q = 1$
12. $Insere((i + m - 1, j), C_O)$
13. $q \leftarrow \bar{\sigma}[q][m - j + 1]$
14. **para** $j \leftarrow 1$ **até** $m - 1$ **faça**
15. $q \leftarrow 0$
16. **para** $i \leftarrow n - m + 2$ **até** n **faça**
17. **enquanto** $q > 0$ **e** $\neg Igual(P[q - 1; m - j + 1, \ldots, m], S[i; 1, \ldots, j])$ **faça**
18. $q \leftarrow \bar{\pi}'[q][j + 1]$
19. **se** $Igual(P[q - 1; m - j + 1, \ldots, m], S[i; 1, \ldots, j])$
20. $q \leftarrow q + 1$
21. **enquanto** $q > 0$ **faça**
22. $Insere((n - q + 1, j), C_{SO})$
23. $q \leftarrow \bar{\pi}'[q][j + 1]$
24. **devolva** C_O e C_{SO}

fim procedimento

Ao final deste procedimento, as posições armazenadas nas listas C_O e C_{SO} são aquelas que servirão para certificar as candidatas das fronteiras Leste e Sudeste do vizinho Oeste. No trecho entre as linhas 4 e 13, são percorridas as posições da faixa vertical de S delimitada pelas $m - 1$ primeiras colunas, da direita para a esquerda e de baixo para cima. Utilizamos a matriz $\bar{\sigma}$ para especificar as linhas do padrão a serem comparadas com as linhas da faixa Oeste. As posições de confirmação obtidas serão armazenadas em C_O e irão certificar as posições da fronteira Leste.

As posições de confirmação para as posições candidatas da fronteira Sudeste, do vizinho Oeste, são obtidas no trecho entre as linhas 14 e 23 deste procedimento, quando percorremos as posições da fronteira Sudoeste da esquerda para a direita e de cima para baixo, utilizando a matriz $\bar{\pi}'$. Ao obtermos as posições de confirmação, estas são incluídas na lista C_{SO}.

No procedimento *Fronteira_ONON* obtemos as posições de certificação para as posições candidatas nas fronteiras: Leste do vizinho Oeste, Sul do vizinho Norte e Sudeste do vizinho Noroeste. A descrição deste procedimento é construída com as descrições dos procedimentos *Fronteira_N* e *Fronteira_O* e a inclusão do trecho a seguir logo após o correspondente no procedimento *Fronteira_ONON* do trecho entre as linhas 4 e 13 de *Fronteira_O*, dentro do laço da linha 4. As posições de confirmação obtidas são armazenadas nas listas C_O, C_{SO}, C_N, C_{NE} e C_{NO}.

$\vdots$
enquanto $q < m + 1$ **faça**
$Insere((m - q + 1, j), C_{NO})$
$q = \bar{\sigma}[q][m - j + 1]$
$\vdots$

Após a obtenção dos candidatos a ocorrência nas fronteiras Sul, Sudeste e Leste, pelo passo 2 do algoritmo *Busca_sem_Escala_Bi_CGM* (página 65), que estão armazenados em C_E, C_S e

C_{SSEE}, conforme os índices dos processadores, estes candidatos deverão ser confirmados pelos dados dos processadores vizinhos. As posições de confirmação obtidas pelo passo 4 e armazenadas nas listas C_N, C_{NE}, C_O, C_{SO} e C_{NO} serão enviadas para seus vizinhos, de acordo com os índices dos processadores onde estão armazenadas.

Os dados recebidos e os dados localmente obtidos são, então, combinados, utilizando-se o procedimento *Combina_Bi*. O comportamento deste procedimento depende do índice do processador e, conseqüentemente, dos dados armazenados. Vamos descrever, a seguir, um trecho que combina os candidados da fronteira Sul, armazenados em C_S, e as posições de confirmação recebidas na lista C_N. As descrições dos trechos para os demais casos são semelhantes e serão omitidas.

PROCEDIMENTO: Combina_Bi
Entrada: as listas C_S, C_E, C_{SSEE}, C_N, C_O, C_{ONON} e R.
Saída: a lista R com as posições que correspondem a ocorrências de P na submatriz de texto S.

⋮

Comentário: o trecho a seguir combina os candidatos da fronteira Sul, armazenados na lista C_S com as posições de confirmação da fronteira Norte, armazenadas na lista C_N. As variáveis c e b são estruturas com campos i e j que contém os valores de linhas e colunas das posições nas listas C_S e C_N.

$R' \leftarrow C_S$
$c \leftarrow Elemento(C_S)$
$b \leftarrow Elemento(C_N)$
enquanto $C_S \neq \emptyset$ **e** $C_N \neq \emptyset$ **faça**
  se $c.j = b.j - m + 1$
    se $n - c.i + b.i + 1 < m$
      $Remove(C_E)$
      $b \leftarrow Elemento(C_N)$
    caso contrário
      se $n - c.i + b.i + 1 > m$
        $Remove(C_S)$
        $d \leftarrow Elemento(C_S)$
        caso contrário
          $C_S \leftarrow Próximo(C_S)$
          $c \leftarrow Elemento(C_S)$
          $Remove(C_N)$
          $b \leftarrow Elemento(C_N)$
  caso contrário
    $aux \leftarrow c.j$
    enquanto $c.j = aux$ **faça**
      $Remove(C_S)$

$c \leftarrow Elemento(C_S)$
enquanto $C_S \neq \emptyset$ **faça**
$Remove(C_S)$
$R \leftarrow Intercala(R, R')$
⋮
devolva R

fim procedimento

3.3.4 Tempo e Corretude

O tempo e a corretude do algoritmo *Busca_sem_Escala_Bi_CGM* (página 65) dependem dos procedimentos envolvidos em sua descrição.

No caso da corretude, as demonstrações são extensões das corretudes dos casos unidimensionais do capítulo anterior ou decorrem da corretude do algoritmo *KMP_2* (página 59).

As análises dos tempos de processamento local, número de rodadas de comunicação e memória utilizada estão descritos a seguir.

Teorema 16 *Os procedimentos Fronteira_S, Fronteira_E e Fronteira_SSEE levam tempo de pior caso* $O(nm)(\leq O(N^2/p))$ *e utilizam memória* $O(n^2)(= O(N^2/p))$ *cada um para determinar as listas de posições candidatas.*

Prova. No procedimento *Fronteira_S*, o laço **para** da linha 4 é executado $n - m + 1$ vezes. Em cada iteração deste laço, o laço **para** da linha 6 é executado $m - 1$ vezes. O corpo deste último laço tem tempo amortizado $O(1)$. Dessa forma, o laço da linha 6 leva tempo $O(m)$. O laço **enquanto** da linha 11 tem tempo de pior caso $O(m)$. Logo, o tempo total do laço da linha 4 é $O(nm) \leq O(N^2/p)$, que também é o tempo do procedimento.

Analogamente, o procedimento *Fronteira_E* também leva tempo $O(nm) \leq O(N^2/p)$. O procedimento *Fronteira_SSEE* é formado pelos procedimentos *Fronteira_S* e *Fronteira_E* e mais um laço **enquanto** de tempo $O(m)$ englobado pelo laço **para** mais externo da obtenção da fronteira Leste. Portanto, este procedimento também leva tempo de pior caso $O(nm) \leq O(N^2/p)$.

Em qualquer um dos procedimentos estarão armazenadas localmente as matrizes S, de tamanho n^2, P, de tamanho m^2, $\bar{\pi}$, de tamanho m^2. Além destas, teremos um número constante de variáveis auxiliares e as listas de candidatos de tamanho $O(nm)$. Desta forma, o espaço utilizado é $O(n^2) = O(N^2/p)$.

□

Teorema 17 *Os procedimentos Fronteira_N, Fronteira_O e Fronteira_ONON levam tempo de pior caso* $O(nm)(\leq O(N^2/p))$ *e utilizam memória* $O(n^2)(= O(N^2/p))$ *cada um para determinar as listas de posições candidatas.*

Prova. No procedimento *Fronteira_N* o laço **para** da linha 4 é repetido $n-m+1$ vezes. Dentro dele o laço **para** da linha 6 executa $m-1$ vezes um corpo de tempo amortizado $O(1)$. Assim, este laço tem tempo $O(m)$. O laço **enquanto** da linha 11 é executado no máximo $m-1$ vezes e está dentro do laço da linha 4. Logo, o laço da linha 4 tem tempo de pior caso $O(nm)$. Por argumentos análogos, o laço **para** da linha 14 leva tempo $O(m^2)$ e é independente do laço da linha 4. Daí temos que o procedimento todo leva tempo $O(nm) \leq O(N^2/p)$ no pior caso.

O procedimento *Fronteira_O* tem também a mesma complexidade de tempo e a justificativa é similar. O procedimento *Fronteira_ONON* é composto por esses dois algoritmos e mais um laço **enquanto** dentro de um laço externo, levando, assim, tempo $O(nm) \leq O(N^2/p)$.

Os três procedimentos armazenam, localmente, a matriz S de tamanho n^2, a matriz P de tamanho m^2, as matrizes $\bar{\sigma}$ e/ou $\bar{\sigma}'$ e/ou $\bar{\pi}'$ de tamanho m^2 cada uma, além de um número constante de variáveis auxiliares e das listas de posições de confirmação de tamanho máximo $O(nm)$. Assim, a memória total utilizada, por cada um destes procedimentos, é $O(n^2) = O(N^2/p)$.

□

O trecho do procedimento *Combina_Bi* descrito leva tempo proporcional à soma dos tamanhos das listas C_S e C_N, que é $O(nm) \leq O(N^2/p)$, por motivos análogos ao procedimento *Combina* (página 37) do capítulo anterior. As listas são percorridas e pelo menos um elemento de cada uma é removido ou caminha-se para o próximo elemento da lista C_S, diminuindo a quantidade de elementos a serem analisados em cada passo. O funcionamento de *Combina_Bi* para quaisquer pares de listas é basicamente o mesmo, levando tempo proporcional à soma dos tamanhos das listas, isto é, o tempo total é no máximo $O(N^2/p)$. A memória utilizada por ele é no máximo $O(n^2) = O(N^2/p)$, pois armazena a lista final com todas as posições de ocorrência.

Teorema 18 *O algoritmo Busca_sem_Escala_Bi_CGM leva tempo* $O(n^2 + m^2)(= O(N^2/p))$, *utilizando memória* $O(n^2)(= O(N^2/p))$ *e* $O(1)$ *rodadas de comunicação, onde são trocados* $O(nm)(\leq O(N^2/p))$ *dados.*

Prova. No passo 1, é executado o algoritmo *KMP_2* que leva tempo $O(n^2 + m^2)$, utilizando memória $O(n^2 + m^2)$.

No passo 2, cada processador, conforme seu índice, obtém as posições candidatas nas fronteiras, levando tempo $O(nm)$ e utilizando $O(n^2)$ de memória.

O passo 3 obtém as matrizes $\bar{\sigma}$, $\bar{\pi}'$ e $\bar{\sigma}'$ em tempo $O(m^2)$, utilizando espaço $O(m^2)$.

O passo 4 é responsável pela obtenção das posições de confirmação nas fronteiras N, NO e O, levando tempo $O(nm)$ e espaço $O(n^2)$.

Em nenhum dos passos descritos é necessária comunicação entre os processadores. O único passo em que se faz comunicação é o passo 5, onde são enviados no máximo $O(nm)$ dados para os processadores vizinhos.

No passo 6, os dados recebidos são combinados com os dados locais, confirmando, ou não, as posições candidatas como ocorrências do padrão no texto. Este passo leva tempo $O(nm)$ e

utiliza espaço $O(n^2)$.

Portanto, o algoritmo leva tempo total $O(n^2 + m^2) = O(N^2/p)$, utilizando memória $O(n^2) = O(N^2/p)$ e uma rodada de comunicação, onde são trocados $O(nm) \leq O(N^2/p)$ dados.

□

Com este resultado, mostramos que o algoritmo tem tempo de computação local $O(N^2/p)$, utiliza um número constante de rodadas de comunicação, onde são trocados no máximo $O(N^2/p)$ dados e o espaço necessário para os dados é também $O(N^2/p)$.

A corretude do procedimento *Fronteira_S* decorre da corretude do procedimento *Fronteira_Esquerda* (página 36) do capítulo anterior, considerando-se cadeias de comprimento m iniciando em cada linha da coluna j como sendo símbolos a serem comparados com os símbolos de $\bar{P}$.

A corretude do procedimento *Fronteira_E* decorre da corretude do procedimento *KMP_2* (página 59), onde o padrão utilizado é uma submatriz do padrão dado.

O procedimento *Fronteira_SSEE*, por sua vez, tem sua descrição baseada nos anteriores e sua corretude é baseada na corretude destes.

Os procedimentos *Fronteira_N* e *Fronteira_O* tem suas corretudes baseadas na corretude do procedimento *Fronteira_Direita* (página 35) do capítulo anterior e da corretude do algoritmo *KMP_2* (página 59). A corretude de *Fronteira_ONON* decorre das anteirores.

Finalmente, o procedimento *Combina_Bi* tem sua corretude decorrente da corretude do procedimento *Combina* (página 37) do capítulo anterior.

3.3.5 Resultados da Implementação

O algoritmo CGM relativo ao algoritmo seqüencial linear para busca bidimensional sem escala foi implementado e foram realizados testes. Os dados experimentais descritos nesta seção foram obtidos usando-se uma matriz texto de ordem $N = 96$ e matrizes padrão de ordem $m = 4$ e $m = 24 (= 96/4)$. Estas matrizes foram formadas apenas pelo símbolo a em todas suas posições e assim as ocorências existiram, globalmente, em todas as posições do texto, exceto nas últimas $m - 1$ linhas e/ou últimas $m - 1$ colunas. Nos testes utilizamos 1, 4 e 16 processadores da máquina *Parsytec PowerXplorer*.

Os tempos absolutos obtidos com estas instâncias estão mostrados na Figura 3.6. Observemos que, para 1 processador, o tempo para $m = 4$ é maior, uma vez que o número de posições a serem analisadas é maior. Para 4 e 16 processadores, o tempo para $m = 4$ é menor uma vez que a quantidade de informações a serem trocadas é menor e a comunicação é a parcela principal no tempo total de execução do algoritmo. É justamente esta quantidade menor de dados trocados que faz com que o tempo caia mais consideravelmente para o padrão de ordem menor. Os valores dos tempos que geraram o gráfico estão no apêndice A.

Na Figura 3.7 observamos que o *speedup* para $m = 4$ é muito próximo do ótimo para $p = 4$ processadores. O *speedup* para $m = 24$ é menos significativo, uma vez que temos o domínio do

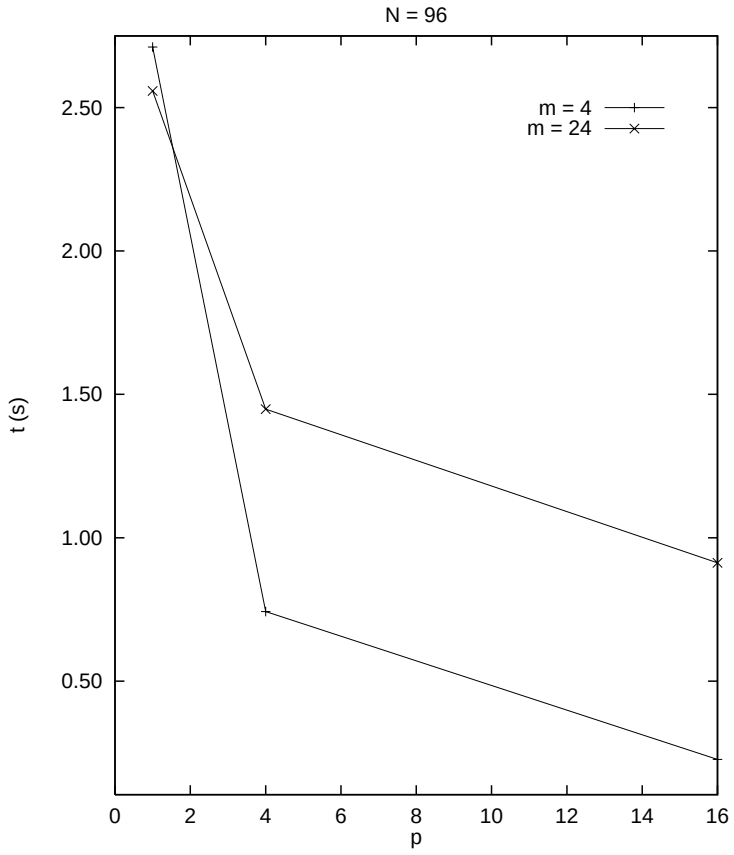

Figura 3.6: Tempo em s para $N = 96$, $m = 4$ e $m = 24$, utilizando 1, 4 e 16 processadores.

tempo de comunicação sobre o tempo de computação.

3.4 Um Algoritmo Seqüencial Sublinear

As idéias envolvidas no algoritmo descrito nesta seção serão utilizadas no desenvolvimento do algoritmo de busca bidimensional de padrões com escala.

A sublinearidade deste algoritmo é atingida, levando-se em conta o formato dos dados de entrada. Para que esta sublinearidade seja obtida, a entrada deve estar em uma forma comprimida e as seguintes formas de compressão devem estar presentes:

- Compressão de símbolos sucessivos iguais dentro de cada linha do padrão e do texto.
- Compressão de sublinhas sucessivas iguais do texto ou do padrão.

Amir *et al* em [8] utilizaram o termo sublinear em relação ao tamanho n^2 da matriz texto de entrada, entretanto como a entrada está na forma fatorada o algoritmo, na realidade, tem

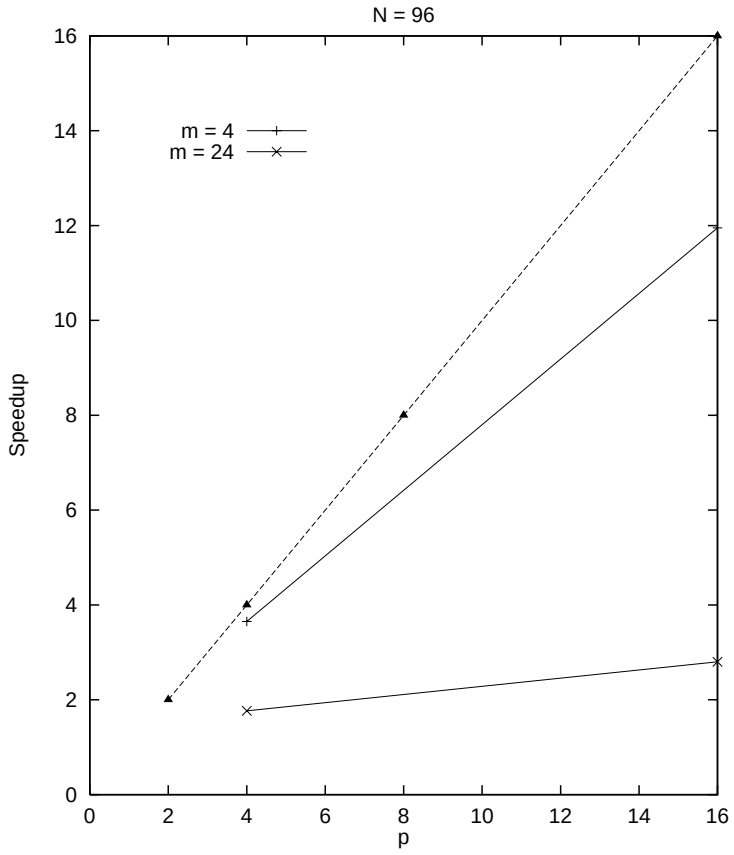

Figura 3.7: *Speedup* para 4 e 16 processadores com $N = 96$, $m = 4$ e $m = 24$.

tempo linear no tamanho da entrada. No decorrer da descrição continuaremos a usar o termo sublinear (em relação ao tamanho n^2).

A sublinearidade do algoritmo depende de quão eficientes serão estas compressões. Por exemplo: se cada linha das matrizes texto e padrão forem formadas por dois símbolos diferentes alternando-se, e cada duas linhas consecutivas forem diferentes não será possível obter a sublinearidade. Isto porque as compressões sobre as matrizes texto e padrão não reduzem a quantidade de dados.

Definição 10 *Uma submatriz* $T' = T[i_1, \ldots, i_1+h; j_1, \ldots, j_1+k-1]$ *de* T *é um* ***bloco*** *na posição* i_1 *da coluna* j_1 *se todas as linhas de* T' *são iguais e nenhuma linha pode ser adicionada a* T' *sem que a condição anterior seja corrompida e* $k = m$. *Se* $0 \leq k < m$ *com as mesmas condições, a submatriz* T' *é denominada* ***sub-bloco de largura*** **k**. *Qualquer linha* i, *com* $i_1 \leq i \leq i_1+h$, *será denominada* ***linha representante do bloco*** *(ou* ***sub-bloco****), ou simplesmente,* ***representante****. A* ***altura*** *do bloco (ou sub-bloco) é o número de linhas* h *do bloco (ou sub-bloco).*

Para que a eficiência do algoritmo seja atingida, agrupamos as n colunas de T em grupos de

m colunas consecutivas. Assim, temos os grupos $\{1,\ldots,m\}$, $\{m+1,\ldots,2m\}$, $\ldots$, $\{(\lfloor\frac{n}{m}\rfloor-1)m+1,\ldots,\lfloor\frac{n}{m}\rfloor m\}$, possivelmente seguido de um grupo com menos de m colunas, $\{\lfloor\frac{n}{m}\rfloor m+1,\ldots,n\}$. Cada grupo de colunas é processado como um todo. Para isto, utilizamos como guia a m-ésima coluna de cada grupo, a qual denominamos *coluna potência* (*power column*).

Este algoritmo é similar ao algoritmo da seção 3.2 apresentando também três etapas. Na primeira etapa, é construída uma estrutura de dados para que comparações do prefixo comum mais longo possam ser feitas em tempo constante. Além disso, a estrutura de dados vai permitir a obtenção de blocos, para cada coluna, em tempo proporcional à quantidade destes. Na segunda etapa, é construído um vetor, a partir da análise do padrão, que é similar ao obtido na análise do padrão da seção 3.2.2. Finalmente, na terceira etapa, a de análise do texto, para cada coluna do texto serão determinados todos os seus blocos e, então, o texto será percorrido através dos blocos em cada coluna.

Exemplo 5(a) *Matriz texto e matriz padrão.*

T

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	a	a	b	a	b	b	b	a	a	b	b	b	a	a	b	b	a	b	b	a
2	a	a	a	a	b	a	b	a	b	b	a	b	b	b	a	a	b	a	a	a
3	a	a	a	a	b	a	b	a	b	a	a	b	b	b	a	a	b	a	a	a
4	b	a	a	a	b	b	a	b	b	a	b	b	a	b	a	b	a	a	b	b
5	b	a	b	b	a	a	b	a	a	a	b	a	b	b	b	b	a	b	b	b
6	b	a	b	b	a	a	b	a	a	a	b	a	a	b	a	b	a	b	b	b
7	b	b	a	b	a	b	a	a	b	b	a	b	b	b	b	a	a	b	b	a
8	b	b	a	b	a	b	a	a	b	b	a	b	b	b	a	b	b	a	a	b
9	a	b	b	a	b	b	a	b	a	a	b	a	b	b	b	b	b	b	a	b
10	a	b	b	a	b	b	a	b	a	a	b	a	a	a	a	a	b	a	b	a
11	b	a	b	a	b	b	a	b	a	a	b	b	b	a	a	a	b	a	b	a
12	b	b	a	b	a	a	b	a	a	a	b	b	b	a	a	b	a	b	b	a
13	b	b	a	b	a	a	b	a	a	a	b	b	b	a	b	b	a	b	b	a
14	a	b	a	a	a	b	b	b	b	a	b	b	b	a	b	b	a	b	b	a
15	b	b	a	a	a	b	b	b	b	a	b	b	b	a	b	a	b	a	a	b
16	a	b	a	a	a	b	b	b	b	a	b	b	b	a	b	a	b	a	a	b
17	a	b	a	a	a	b	a	a	b	b	a	b	b	a	b	b	b	a	a	b
18	a	b	a	a	a	b	b	b	a	a	b	a	a	a	b	a	a	a	b	b
19	a	b	a	a	a	b	b	b	a	a	b	a	a	a	b	a	b	b	a	a
20	b	b	a	b	b	b	a	b	a	b	a	a	b	b	a	b	b	b	a	a

P

	1	2	3	4	5	6	7	8
1	a	a	b	b	a	b	b	a
2	b	b	a	a	b	a	a	a
3	b	b	a	a	b	a	a	a
4	a	b	a	b	a	a	b	b
5	a	b	a	b	a	a	b	b
6	b	a	b	b	a	b	a	a
7	b	a	b	b	a	b	a	a
8	b	a	b	b	a	b	a	a

A entrada do algoritmo será constituída pela forma fatorada FT da matriz texto T e pela matriz fatorada FP da matriz padrão P. Ambas FT e FP são dadas na representação fatorada de suas linhas e cada símbolo destas matrizes é um par ordenado de símbolo e expoente.

Podemos acessar estes valores separadamente, considerando estas matrizes como estruturas com campos símbolo e expoente. No exemplo 5(a) temos um exemplo de uma entrada de dados; no exemplo 5(b), as representações fatoradas destas matrizes linha a linha, com os símbolos (S) e os expoentes (E). Os dados destes exemplos serão usados nos próximos exemplos para ilustrar o funcionamento dos passos seguintes.

Exemplo 5(b) *As representações fatoradas FT e FP de T e P, respectivamente, linha a linha, com os valores dos símbolos em S e dos expoentes em E.*

$FT[1]$

										10	11
C	1	2	3	4	5	6	7	8	9	0	1
S	a	b	a	b	a	b	a	b	a	b	a
E	2	1	1	3	2	3	2	2	1	2	1

$FT[2]$

									20	21	22
	12	13	14	15	16	17	18	19			
C	2	3	4	5	6	7	8	9	0	1	2
S	a	b	a	b	a	b	a	b	a	b	a
E	4	1	1	1	1	2	1	3	2	1	3

$FT[3]$

	22	22	22	22	22	22	23	30	31	32	33
C	3	4	5	6	7	8	9	0	1	2	3
S	a	b	a	b	a	b	a	b	a	b	a
E	4	1	1	1	1	1	2	3	2	1	3

$FT[4]$

	33	33	33	33	33	34	40	41	42	43	44	45	46
C	4	5	6	7	8	9	0	1	2	3	4	5	6
S	b	a	b	a	b	a	b	a	b	a	b	a	b
E	1	3	2	1	2	1	2	1	1	1	1	2	2

$FT[5]$

	44	44	44	45	45	45	45	45	45	45	45
C	7	8	9	0	1	2	3	4	5	6	7
S	b	a	b	a	b	a	b	a	b	a	b
E	1	1	2	2	1	3	1	1	4	1	3

$FT[6]$

	55	55	56	56	56	56	56	56	56	56	56	56	57
C	8	9	0	1	2	3	4	5	6	7	8	9	0
S	b	a	b	a	b	a	b	a	b	a	b	a	b
E	1	1	2	2	1	3	1	2	1	1	1	1	3

$FT[7]$

	77	77	77	77	77	77	77	77	77	78	78	78
C	1	2	3	4	5	6	7	8	9	0	1	2
S	b	a	b	a	b	a	b	a	b	a	b	a
E	2	1	1	1	1	2	2	1	4	2	2	1

$FT[8]$

	88	88	88	88	88	88	88	89	89	89	89	89	89
C	3	4	5	6	7	8	9	0	1	2	3	4	5
S	b	a	b	a	b	a	b	a	b	a	b	a	b
E	2	1	1	1	1	2	2	1	3	1	2	2	1

$FT[9]$

					10	10	10	10	10	10	10	10
	99	99	99	99	00	00	00	00	00	00	00	00
C	6	7	8	9	0	1	2	3	4	5	6	7
S	a	b	a	b	a	b	a	b	a	b	a	b
E	1	2	1	2	1	1	2	1	1	5	2	1

$FT[10]$

	11	11	11	11	11	11	11	11	11	11	11	11	11
	00	00	11	11	11	11	11	11	11	11	11	11	12
C	8	9	0	1	2	3	4	5	6	7	8	9	0
S	a	b	a	b	a	b	a	b	a	b	a	b	a
E	1	2	1	2	1	1	2	1	5	1	1	1	1

$FT[11]$

	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
	22	22	22	22	22	22	22	22	22	23	23	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24
C	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7
S	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b
E	1	1	1	1	2	1	1	2	3	3	1	1	1	1	2	1	1	2	1	3	3	2	1	1	2	1	1

$FT[12]$

	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
	22	22	22	22	22	22	22	22	22	23	23	23	23	23	23	23	23	23	23	24	24	24	24	24	24	24	24
C	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7
S	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a	b
E	1	1	1	1	2	1	1	2	3	3	1	1	1	1	2	1	1	2	1	3	3	2	1	1	2	1	1

$FT[13]$

	11	11	11	11	11	11	11	11	11	11	11	11
	44	44	44	45	45	45	45	45	45	45	45	45
C	7	8	9	0	1	2	3	4	5	6	7	8
S	b	a	b	a	b	a	b	a	b	a	b	a
E	2	1	1	2	1	3	3	1	2	1	2	1

$FT[14]$

	11	11	11	11	11	11	11	11	11	11	11	11
	55	56	56	56	56	56	56	56	56	56	56	56
C	9	0	1	2	3	4	5	6	7	8	9	
S	a	b	a	b	a	b	a	b	a	b	a	
E	1	1	3	4	1	3	1	2	1	2	1	

$FT[15]$

	1	1	1	1	1	1	1	1	1	1	1
	7	7	7	7	7	7	7	7	7	7	8
C	0	1	2	3	4	5	6	7	8	9	0
S	b	a	b	a	b	a	b	a	b	a	b
E	2	3	4	1	3	1	1	1	1	2	1

$FT[16]$

	1	1	1	1	1	1	1	1	1	1	1
	8	8	8	8	8	8	8	8	8	9	9
C	1	2	3	4	5	6	7	8	9	0	1
S	a	b	a	b	a	b	a	b	a	b	a
E	1	1	3	4	1	3	1	1	1	1	2

$FT[17]$

	1	1	1	1	1	1	1	2	2	2	2
	9	9	9	9	9	9	9	0	0	0	0
C	3	4	5	6	7	8	9	0	1	2	3
S	a	b	a	b	a	b	a	b	a	b	a
E	1	1	3	1	2	2	1	2	1	3	2

$FT[18]$

	2	2	2	2	2	2	2	2	2	2
	0	0	0	0	0	1	1	1	1	1
C	5	6	7	8	9	0	1	2	3	4
S	a	b	a	b	a	b	a	b	a	b
E	1	1	3	3	2	1	3	1	3	2

$FT[19]$

	2	2	2	2	2	2	2	2	2	2	2
	1	1	1	1	1	2	2	2	2	2	2
C	5	6	7	8	9	0	1	2	3	4	5
S	a	b	a	b	a	b	a	b	a	b	a
E	1	1	3	3	2	1	3	1	1	2	2

$FT[20]$

	2	2	2	2	2	2	2	2	2	2	2	2
	2	2	2	2	3	3	3	3	3	3	3	3
C	6	7	8	9	0	1	2	3	4	5	6	7
S	b	a	b	a	b	a	b	a	b	a	b	a
E	2	1	3	1	1	1	1	2	2	1	3	2

$FP[1]$

	2	2	2	2	2
	3	3	4	4	4
C	8	9	0	1	2
S	a	b	a	b	a
E	2	2	1	2	1

$FP[2]$

	2	2	2	2
	4	4	4	4
C	3	4	5	6
S	b	a	b	a
E	2	2	1	3

$FP[3]$

	2	2	2	2
	4	4	4	5
C	7	8	9	0
S	b	a	b	a
E	2	2	1	3

$FP[4]$

	2	2	2	2	2	2
	5	5	5	5	5	5
C	1	2	3	4	5	6
S	a	b	a	b	a	b
E	1	1	1	1	2	2

$FP[5]$

	2	2	2	2	2	2
	5	5	5	6	6	6
C	7	8	9	0	1	2
S	a	b	a	b	a	b
E	1	1	1	1	2	2

$FP[6]$

	2	2	2	2	2	2
	6	6	6	6	6	6
C	3	4	5	6	7	8
S	b	a	b	a	b	a
E	1	1	2	1	1	2

$FP[7]$

	2	2	2	2	2	2
	6	7	7	7	7	7
C	9	0	1	2	3	4
S	b	a	b	a	b	a
E	1	1	2	1	1	2

$FP[8]$

	2	2	2	2	2	2
	7	7	7	7	7	8
C	5	6	7	8	9	0
S	b	a	b	a	b	a
E	1	1	2	1	1	2

3.4.1 Pré-Processamento da Entrada

Nesta etapa, forma-se uma cadeia C obtida pela concatenação de todas as linhas de FT seguida pela concatenação de todas as linhas de FP. A partir de C constrói-se uma árvore de sufixos ST de C. Na construção desta árvore, considera-se cada par símbolo e expoente como sendo um símbolo. A partir daqui, o único objeto contendo os dados de entrada na forma fatorada a que teremos acesso será esta cadeia C. Para mantermos uma referência das linhas de FT e FP, teremos os vetores Fim_FT e Fim_FP de comprimento $n+1$ e $m+1$, respectivamente, com os valores dos índices em C do último elemento de cada linha de FT e FP. Vamos considerar que $Fim\_FT[0] = 0$ e que $Fim\_FP[0] = Fim\_FT[n]$.

Aplica-se à árvore ST um algoritmo para o LCA. Com isto, o prefixo comum mais longo pode ser obtido em tempo constante, conforme visto na seção 1.9.

Além disso, uma outra estrutura de dados se faz necessária para, dada uma coluna potência c e uma outra coluna $j \geq c$ (alternativamente, $j < c$), encontrar todas as linhas $1 \leq i \leq n-1$, tais que as linhas i e $i+1$ sejam diferentes entre as colunas c e j (alternativamente, entre as colunas j e $c-1$). Ou seja, $T[i; c, \ldots, j] \neq T[i+1; c, \ldots, j]$ (alternativamente, $T[i; j, \ldots, c-1] \neq T[i+1; j, \ldots, c-1]$).

Seja $[i, c]$ uma posição de uma coluna potência c em T. Seja $\mathbf{B_r[i, c]}$ o maior inteiro k para o qual as cadeias $T[i; c, c+1, \ldots, c+k-1]$ e $T[i+1; c, c+1, \ldots, c+k-1]$ sejam iguais. Analogamente, seja $\mathbf{B_l[i, c]}$ o maior inteiro k para o qual as cadeias $T[i; c-k, c-k+1, \ldots, c-1]$ e $T[i+1; c-k, c-k+1, \ldots, c-1]$ sejam iguais. Isto é, $B_r[i, c]$ nos dá o comprimento do prefixo comum mais longo das linhas i e $i+1$ começando na coluna c e $B_l[i, c]$ nos dá o comprimento do sufixo comum mais longo das linhas i e $i+1$ terminando na coluna $c-1$.

Os valores de $B_r[i, c]$ para cada coluna potência c são obtidos percorrendo-se cada par de linhas i e $i+1$ de FT, da direita para a esquerda, determinando o valor de $B_r[i, c]$ cada vez que uma coluna potência é encontrada. O número de operações é proporcional à soma do comprimento fatorizado das linhas i e $i+1$ e o número de colunas potência, $\lfloor \frac{n}{m} \rfloor$. Os valores de $B_l[i, c]$ para cada coluna potência c são obtidos analogamente, percorrendo-se os pares de linhas da esquerda para a direita. No exemplo 5(c) temos os vetores B_l e B_r construídos para as colunas potência 8 e 16.

Exemplo 5(c) *Vetores B_r e B_l para as colunas potência 8 e 16.*

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$B_r[8]$	1	2	0	0	5	1	7	0	5	4	0	7	0	8	13	0	0	9	2
$B_r[16]$	0	5	0	2	5	0	0	5	0	5	0	5	5	0	5	0	0	1	0
$B_l[8]$	1	7	0	0	7	0	7	2	7	5	0	7	1	6	6	0	0	7	0
$B_l[16]$	0	5	2	0	0	0	0	0	0	2	7	0	6	14	14	4	2	15	0

Para cada coluna potência c consideramos os vetores $B_r[1, \ldots, n-1; c]$ e $B_l[1, \ldots n-1; c]$ e construímos as árvores Cartesianas CT_r e CT_l para estes vetores, respectivamente. Estas estruturas serão utilizadas na etapa de análise do texto.

3.4.2 Análise do Padrão

Sejam $P_1, P_2, \ldots, P_m$ as linhas da matriz padrão P e $FP_1, FP_2, \ldots, FP_m$ as respectivas representações fatoradas destas linhas. Se considerarmos cada uma destas representações fatoradas FP_i como sendo um símbolo, teremos a seqüência $FP_1, FP_2, \ldots, FP_m$ vista como uma cadeia unidimensional R. Por outro lado, podemos descrever FP como a linha FP_{l_1} repetida r_1 vezes (o bloco na linha l_1), seguida pela linha $FP_{l_2} (\neq FP_{l_1})$ repetida r_2 vezes, seguida por outros blocos até que, finalmente, tenhamos a linha FP_{l_α} $(\neq FP_{l_{\alpha-1}})$ repetida r_α vezes, onde $l_1 = 1$, $l_k = r_1 + r_2 + \cdots + r_{k-1}$ e $r_1 + r_2 + \cdots + r_\alpha = m$. Formalmente, um par (FP_{l_k}, r_k) é um bloco de r_k linhas na linha l_k. Assim, a representação fatorada de R, FR, é dada por esta seqüência de pares.

Nesta etapa, inicialmente comprimimos a cadeia R em sua representação fatorada, denotada por FR. Isto leva tempo proporcional a $|FP_1| + |FP_2| + \cdots + |FP_m|$.

Vamos supor que $\alpha > 2$, isto é existem pelo menos 3 blocos em FR. Para $\alpha \leq 2$ existe uma solução com a mesma complexidade. Seja $\hat{FR}$ a subseqüência de FR que se inicia no segundo elemento e termina no penúltimo elemento. Isto é, $\hat{FR} = (FP_{l_2}, r_2) \ldots (FP_{l_{\alpha-1}}, r_{\alpha-1})$.

A seguir, calcula-se o vetor π para a cadeia $\hat{F}R$, usando a árvore ST. Esta construção leva tempo $O(\alpha)$.

3.4.3 Análise do Texto

A etapa de análise de texto envolve alguns processamentos antes da análise propriamente dita e estes serão descritos a seguir.

Construção de listas

No primeiro passo desta etapa, para cada coluna j, é construída uma lista ligada $Q[j]$ dos índices de linhas onde os blocos da coluna j terminam. Estas listas são construídas observando-se o seguinte:

Observação 3 *Um bloco na coluna j termina em uma linha i se, e somente se, pelo menos uma das duas condições seguintes ocorre:*

1. $B_l[i,c] < c - j$, *ou*
2. $B_r[i,c] < m - c + j$.

Se $B_l[i,c] \geq c - j$ e $B_r[i,c] \geq m - c + j$, as linhas i e $i+1$ de T, a partir da coluna j, serão iguais em pelo menos m posições e, conseqüentemente, esta linha i não será o fim de um bloco.

Se $B_l[i,c] < c - j$ então $T[i,j] \neq T[i+1,j]$ e, conseqüentemente, a linha i é o fim de um bloco na coluna j. Por outro lado, se $B_r[i,c] < m - c + j$ então $T[i, c + B_r[i,c]] \neq T[i+1, c + B_r[i,c]]$ e, mesmo se $T[i; j, \ldots, j+m-2] = T[i+1; j, \ldots, j+m-2]$, a linha i será o fim de um bloco na coluna j, pois as linhas i e $i+1$ coincidem, a partir da posição j, em menos de m posições consecutivas.

Esses valores são obtidos nas árvores Cartesianas construídas na etapa de pré-processamento (seção 3.4.1). Assim, para cada coluna j e a respectiva coluna potência c do grupo a qual ela pertence, vamos obter a lista de linhas i, em ordem crescente, tais que $B_l[i,c] < c - j$, da árvore Cartesiana $CT_l[c]$ e a lista de linhas i tais que $B_r[i,c] < m - c + j$, da árvore Cartesiana $CT_r[c]$. A partir da intercalação destas listas, para cada coluna j, obtemos as listas $Q[j]$ de linhas nas quais blocos na coluna j terminam.

A construção da lista $Q[j]$, para cada coluna j, leva tempo proporcional a $|Q[j]|$, pois esta é obtida percorrendo-se os nós das árvores Cartesianas até que seja encontrado um valor maior ou igual a $c - j$, em $CT_l[c]$, ou maior ou igual a $m - c + j$, em $CT_r[c]$. No exemplo 5(d) temos algumas listas $Q[j]$ construídas.

Exemplo 5(d) *Listas de fim de bloco para cada coluna $j = \min(n - m + 1, \lfloor \frac{n}{m} \rfloor m)$ (=13).*

$Q[1] = 1 \to 3 \to 4 \to 6 \to 8 \to 10 \to 11 \to 13 \to 14 \to 15 \to 16 \to 17 \to 19 \to 20$
$Q[2] = 1 \to 3 \to 4 \to 6 \to 8 \to 10 \to 11 \to 13 \to 16 \to 17 \to 19 \to 20$

$Q[3] = \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 11 \rightarrow 13 \rightarrow 16 \rightarrow 17 \rightarrow 19 \rightarrow 20$
$Q[4] = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 11 \rightarrow 13 \rightarrow 16 \rightarrow 17 \rightarrow 19 \rightarrow 20$
$Q[5] = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 13 \rightarrow 16 \rightarrow 17 \rightarrow 19 \rightarrow 20$
$Q[6] = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 13 \rightarrow 16 \rightarrow 17 \rightarrow 19 \rightarrow 20$
$Q[7] = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 13 \rightarrow 16 \rightarrow 17 \rightarrow 19 \rightarrow 20$
$Q[8] = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 16 \rightarrow 17 \rightarrow 19 \rightarrow 20$
$Q[9] = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 19 \rightarrow 20$
$Q[10] = 1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 11 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 18 \rightarrow 19 \rightarrow 20$
$Q[11] = 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 18 \rightarrow 19 \rightarrow 20$
$Q[12] = 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 18 \rightarrow 19 \rightarrow 20$
$Q[13] = 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 14 \rightarrow 16 \rightarrow 17 \rightarrow 18 \rightarrow 19 \rightarrow 20$

O procedimento *Compara*

Como vimos no algoritmo linear, nesta etapa precisamos comparar subcadeias da matriz texto com linhas da matriz padrão. Utilizamos naquele caso um procedimento *Igual*. Neste algoritmo, um pré-processamento adicional para esta comparação faz-se necessário e utilizaremos um procedimento *Compara* para comparar um bloco de FP e um bloco de FT.

Este procedimento tem como entrada o bloco em $T[i, j]$ e o bloco na linha l_k de P, representado pelo par (FP_{l_k}, r_k). Temos a garantia que a linha i está em $Q[j]$, portanto é a linha final de um bloco na coluna j, e que o número de linhas do bloco é $i - ant(i)$, onde $ant(i)$ é o valor da linha do antecessor da linha i em $Q[j]$.

A saída do procedimento é 1 se $T[i; j, \ldots, j+m-1] = P_{l_k}$ e $i - ant(i) = r_k$; e 0 caso contrário. Este procedimento obtém a resposta em tempo constante.

Neste procedimento, temos a comparação de duas cadeias e para isto utilizamos uma consulta LCA na árvore de sufixos ST. Entretanto, existe uma dificuldade por causa da forma como a árvore de sufixos é construída. A construção da árvore ST é baseada na representação fatorada das matrizes texto e padrão. Com isto, se $T[i, j-1] = T[i, j]$ o sufixo iniciando em $T[i, j]$ não estará representado na árvore de sufixos ST.

Para contornar este problema temos, para cada tal posição $[i, j]$, o menor $j_1 > j$ para o qual o sufixo começando em $T[i, j_1]$ aparece na árvore de sufixos ST. O valor de j_1 para cada posição i, j e o correspondente índice em C de $T[i, j_1]$ são determinados através do procedimento a seguir. Observemos que as listas $Q[j]$ contêm todas as posições $[i, j]$ que são o fim de um bloco na coluna j, com os valores de i armazenados no campo linha de $Q[j]$, e em cada uma delas o procedimento *Compara* é aplicado. Estas listas estão ordenadas lexicograficamente com j primeiro e i segundo. Para obter os valores de j_1 precisamos reordenar estas listas lexicograficamente com i primeiro e j segundo. Os valores de j_1, para cada bloco em cada lista $Q[j]$, são armazenados no campo j_1 de $Q[j]$. O campo *expoente* de $C[j]$ armazena o expoente do símbolo $C[j]$ na cadeia C. Os conteúdos dos demais campos das listas Q e L estão descritos logo após o algoritmo.

PROCEDIMENTO: Calcula_j_1
Entrada: as listas $Q[j]$
Saída: os valores de j_1 para cada posição $[i, j]$ de $Q[j]$.

1. **para** $j \leftarrow 1$ **até** n **faça**
2. $\quad L[j] = \emptyset$
3. **para** $j \leftarrow 1$ **até** n **faça**
4. $\quad p \leftarrow Q[j]$
5. $\quad$ **enquanto** $p \neq \emptyset$ **faça**
6. $\quad\quad Insere((j, p), L[p.linha])$
7. $\quad\quad p \leftarrow Próximo(p)$
8. $j \leftarrow 1$
9. **para** $i \leftarrow 1$ **até** n **faça**
10. $\quad p \leftarrow L[i]$
11. $\quad cont \leftarrow C[j].expoente$
12. $\quad$ **enquanto** $j \neq Fim\_FT[i]$ **faça**
13. $\quad\quad$ **enquanto** $p \neq \emptyset$ **e** $p.coluna <= cont$ **faça**
14. $\quad\quad\quad (p.pont).j_1 \leftarrow cont + 1$
15. $\quad\quad\quad (p.pont).coluna \leftarrow j + 1$
16. $\quad\quad\quad p \leftarrow Próximo(p)$
17. $\quad\quad$ **se** $p = \emptyset$
18. $\quad\quad\quad j \leftarrow Fim\_FT[i] + 1$
19. $\quad\quad$ **caso contrário**
20. $\quad\quad\quad j \leftarrow j + 1$
21. $\quad\quad\quad cont \leftarrow cont + C[j].expoente$

fim procedimento

Nas linhas 1 e 2 deste procedimento, inicializamos n listas L. Estas listas serão indexadas pelo índice de linha e servirão para armazenar, para cada linha i, as colunas j para as quais i pertence a $Q[j]$. Isto é, estas listas conterão as mesmas posições $[i, j]$ das listas $Q[j]$ porém ordenadas lexicograficamente com i primeiro e j segundo. As linhas 3 a 7 constroem estas listas L, percorrendo as listas $Q[j]$ e inserindo j na lista $L[i]$, onde i é uma linha encontrada em $Q[j]$. Nas listas L também são incluídos para cada posição $[i, j]$, um apontador para o nó de $Q[j]$ que contém i. No trecho entre as linhas 9 e 20, percorremos cada posição $[i, j]$ das listas L, obtendo os valores j_1 que são armazenados no campo j_1 do nó em $Q[j]$ contendo i e o correspondente valor do índice de $T[i, j_1]$ em C, armazenado no campo *coluna*.

Após este procedimento, cada nó das listas $Q[j]$ contém: o valor i da linha final de um bloco, o valor de j_1 em relação à posição $[i, j]$, a posição em C correspondente à posição $[i, j_1]$, e os apontadores para os nós anterior e posterior na lista.

Com os valores de j_1 já obtidos, a comparação entre duas cadeias é dada pelo procedimento a seguir, onde o procedimento *Comum* obtém o comprimento do prefixo comum mais longo dos argumentos passados. O procedimento descrito recebe como argumentos: i, o índice da primeira linha do bloco em T; k o índice da primeira linha do bloco em P; j, a coluna onde o bloco em T começa; um apontador q para o nó da lista $Q[j]$ cujo bloco contém a linha i; e a *altura* do

bloco em P começando na linha l_k.

PROCEDIMENTO: Compara
Entrada: i, o índice da primeira linha do bloco em T; k o índice da primeira linha do bloco em P; j, a coluna onde o bloco em T começa; um apontador q para o nó da lista $Q[j]$ cujo bloco contém a linha i; e a *altura* do bloco em P começando na linha l_k.
Saída: os valores 0 ou 1 se as cadeias forem diferentes ou iguais, respectivamente.
1. $j_1 \leftarrow q.j_1$
2. **se** $j_1 - j = C[Fim\_FP[k-1]+1].expoente$
3. **se** $Comum(T[i; j_1, \ldots], P_{l_k}[j_1 - j + 1, \ldots]) = m - (j_1 - j)$ **e**
$T[i,j] = P[l_k, 1]$ **e** $q.linha - i + 1 = altura$
4. **devolva** 1
5. **devolva** 0

fim procedimento

Devemos observar que no teste da linha 3, fazemos uma referência explícita a posições de T e P. Na realidade, dispomos apenas da cadeia C. Porém, com o valor de j_1 e dos vetores Fim_FT e Fim_FP, podemos obter estas posições. Além disso, em uma implementação devemos tomar cuidado com o último símbolo da linha l_k de FP que pode ter expoente menor que o último símbolo do bloco em T começando na posição $[i, j]$.

Busca do Padrão

Vamos descrever como a análise do texto é executada. Percorremos cada coluna j de T, separadamente, como se fosse uma cadeia unidimensional, comparando os blocos que se iniciam nas linhas de cada uma destas colunas com blocos de P. Para avançar de uma posição analisada para a próxima, utilizamos a lista $Q[j]$ que fornece o próximo fim de bloco.

PROCEDIMENTO: Busca_$\hat{FR}$
Entrada: a cadeia C e a seqüência $\hat{FR}$.
Saída: uma lista R das posições em T onde $\hat{FR}$ ocorre.
{ $\hat{FR} = (FP_{l_2}, r_2) \ldots (FP_{l_{\alpha-1}}, r_{\alpha-1})$. }
1. **para** $j \leftarrow 1$ **até** $n - m + 1$ **faça**
2. $linha[0] \leftarrow 0$
3. $q \leftarrow Q[j]$
4. $i \leftarrow q.linha + 1$
5. $k \leftarrow 1$
6. $cont \leftarrow 1$
7. $linha[cont] \leftarrow q.linha$
8. **enquanto** $i < n$ **e** $\neg(Próximo(q) = fim)$ **faça**
9. **enquanto** $k > 1$ **e** $\neg Compara(i, k+1, j, Próximo(q), r_{k+1})$ **faça**
10. $k \leftarrow \pi[k]$

11. **se** $Compara(i, k+1, j, Próximo(q), r_{k+1})$
12. $k \leftarrow k+1$
13. $cont \leftarrow cont+1$
14. $q \leftarrow Próximo(q)$
15. $i \leftarrow q.linha+1$
16. $linha[cont] \leftarrow q.linha$
17 **se** $k = \alpha - 1$
18. $Insere((linha[cont-k+1] - r_1 + 1, j), R)$
19. $k \leftarrow \pi[k]$
20. **devolva** R

fim procedimento

No procedimento, percorremos os blocos do texto na coluna j, com $1 \leq j \leq n-m+1$, guiados pelas linhas armazenadas na lista $Q[j]$. Para podermos recuperar o início de uma ocorrência, sem que seja necessário percorrer a lista $Q[j]$ novamente, precisamos de um vetor auxiliar $linha$ que armazena as linhas finais dos blocos visitados. Este vetor é indexado pela variável *cont*. A variável q percorre os nós armazenados em $Q[j]$. A variável i armazena a primeira linha do bloco subseqüente ao bloco apontado por q. A variável k percorre os índices de pares da seqüência $\hat{F}R$.

No trecho entre as linhas 3 e 7, é feita a inicialização das variáveis envolvidas. O laço **enquanto** da linha 8 controla o percurso por entre as linhas finais dos blocos em cada coluna j. O trecho entre as linhs 9 e 12 tem funcionamento análogo ao algoritmo *KMP_2* (página 59) e utiliza *Compara* para comparar os blocos do texto e do padrão. Nas linhas 13 a 16, passa-se para o próximo bloco atualizando-se também as variáveis auxiliares.

Devemos notar que o valor inicial de k é 1 e o bloco inicial do texto sendo comparado é o segundo, pois estamos verificando se existe uma ocorrência de $\hat{F}R$ em FP. Esta busca termina quando o teste da linha 17 é verdadeiro e temos esta ocorrência. Então, inserimos a posição de ocorrência na lista R e passamos a buscar uma próxima ocorrência, se ainda for possível.

Para a determinação das ocorrências de P em T, utilizamos este procedimento e, para cada ocorrência de $\hat{F}R$ em T, verificamos se ela se estende a uma ocorrência do padrão todo. Para isto, comparamos os blocos de texto antes e depois da ocorrência com (FP_{l_1}, r_1) e $(FP_{l_\alpha}, r_\alpha)$, respectivamente, verificando se suas representantes são iguais às respectivas representantes do primeiro e último blocos de FR e se as alturas dos blocos de texto são maiores ou iguais a r_1 e r_α, respectivamente. Se o resultado de ambas as comparações for afirmativo, temos uma ocorrência do padrão no texto, e a posição é atualizada em R para conter a posição da ocorrência de P em T; caso contrário, é removida. Assim, a lista R contém as posições de ocorrência de T em P. Em uma implementação, as verificações das extensões podem ser feitas a cada ocorrência de $\hat{F}R$ em T, quando podemos utilizar os dados já disponíveis.

3.4.4 Tempo e Corretude

A corretude do procedimento *Busca_$\hat{F}R$* está baseada na corretude do algoritmo *KMP* (página 25). Para cada coluna j da matriz texto, consideramos cada bloco começando nesta coluna como um símbolo que é comparado com um símbolo de $\hat{F}R$. Estes símbolos da coluna j são formados pela subcadeia começando na última linha do bloco, com comprimento m, e pela altura do bloco, dedutível da lista $Q[j]$. A seqüência de passos para comparar blocos de T e de $\hat{F}R$ é a mesma.

A garantia para o funcionamento correto do procedimento baseia-se na observação de que todos os finais de blocos, para cada coluna j, estão corretamente listados em $Q[j]$. Isto decorre da observação 3.

Para cada ocorrência de $\hat{F}R$ encontrada, compara-se os blocos de T anterior e posterior à ocorrência para verificar se existe uma ocorrência de FR em T. Com isto, todas as ocorrências são obtidas e nenhuma posição de falsa ocorrência estará presente na lista R.

O comprimento da cadeia C é $|FT| + |FP|$. Na seção 1.6 mostramos que é possível construir uma árvore de sufixos para uma cadeia S em tempo $O(|S|)$ usando espaço também $O(|S|)$. Estes algoritmos supõem que os símbolos da cadeia pertençam a um alfabeto de tamanho fixo. No nosso caso, a cadeia C é formada por pares (s, r) onde os símbolos s estão em um alfabeto de tamanho fixo, mas o valor de r está no intervalo $[1, n]$. Lembremos que r é o número de repetições de s na representação não comprimida e que r tem $\log r < r$ *bits*. Sejam C_b a concatenação das seqüências $sr_1r_2 \ldots r_{\log r}$ onde r_i é o i-ésimo *bit* na representação binária de r, FT_b a parte de C_b induzida por FT e FP_b a parte de C_b induzida por FP.

Temos, $|C| \leq |C_b| = |FT_b| + |FP_b| \leq |T| + |P|$. C_b é uma cadeia sobre um alfabeto de tamanho fixo, logo a árvore de sufixos para ela pode ser construída em tempo $O(|C_b|)$.

Como a árvore de sufixos utiliza espaço $O(|C_b|)$, o algoritmo para o problema de LCA nesta árvore leva tempo $O(|C_b|)$.

A obtenção dos vetores B_r e B_l leva tempo

$$\sum_{i=1}^{n-1} |FT_i| + |FT_{i-1}| + \lfloor \frac{n}{m} \rfloor = O(\lfloor \frac{n^2}{m} \rfloor + |FT|).$$

A construção das árvores Cartesianas para cada um dos vetores B_r e B_l leva tempo linear no tamanho dos vetores. Como são $\lfloor \frac{n}{m} \rfloor$ vetores B_r e $\lfloor \frac{n}{m} \rfloor$ vetores B_l de comprimento $n-1$ cada um, estas árvores são obtidas em tempo total $O(\lfloor \frac{n^2}{m} \rfloor)$.

A compressão da cadeia R, na seção 3.4.2, leva tempo proporcional a $|FP|$ e a obtenção do vetor π para $\hat{F}R$ leva tempo $O(|\hat{F}R|) \leq O(m)$.

A obtenção das listas $Q[j]$ para cada coluna j leva tempo proporcional ao número de blocos em cada coluna, que é justamente o número de nós de cada lista, ou seja, estas listas são obtidas em tempo $O(\sum_{j=1}^{n} |Q[j]|)$.

O pré-processamento adicional leva tempo $O(|FT| + \sum_{j=1}^{n} |Q[j]|)$, para se obter os valores de j_1.

Finalmente, o procedimento *Busca_*$\hat{FR}$ leva tempo $O(\sum_{j=1}^{n} |Q[j]|)$ e a verificação das extensões leva tempo constante.

Assim, a determinação das ocorrências é feita em tempo $O(\lfloor \frac{n^2}{m} \rfloor + |C_b| + \sum_{j=1}^{n} |Q[j]|)$. Temos que $\lfloor \frac{n^2}{m} \rfloor < n^2$ e que $|C_b| \leq |T| + |P|$. A sublinearidade do algoritmo depende da entrada, por exemplo se o texto e o padrão são formados por um único símbolo, temos $|C_b| = (n+m) \log m < n^2$ e $\sum_{j=1}^{n} |Q[j]| = n < n^2$.

3.5 Algoritmo CGM Baseado no Algoritmo Seqüencial Sublinear

A entrada deste algoritmo será uma matriz de texto de ordem N e uma matriz padrão de ordem m. Os dados serão distribuídos entre os processadores da mesma maneira da seção 3.3.1, onde cada processador recebe uma submatriz de texto S de ordem $n = N/\sqrt{p}$ e a matriz padrão P de ordem m, e supomos $m \leq n/\sqrt{p}$. Vamos utilizar no algoritmo as representações fatoradas FS e FP de S e P, respectivamente.

Os conceitos de vizinhança entre processadores e submatrizes, bem como os conceitos de fronteiras, serão os mesmos descritos na seção 3.3.1.

A sublinearidade do algoritmo seqüencial não será mantida nas computações locais por causa dos dados nas regiões de fronteira que exigem mais informações a serem obtidas na análise do padrão, como veremos a seguir. O algoritmo apresenta tempo de computação local $O(n^2+m^2) = O(N^2/p)$, consumindo memória $O(n^2 + m^2) = O(N^2/p)$ e utilizando apenas uma rodada de comunicação, onde são trocados no máximo $O(nm) \leq O(N^2/p)$ dados. Esta versão CGM é importante na obtenção de conceitos e idéias que serão utilizados na descrição do algoritmo CGM para o caso bidimensional com escala.

3.5.1 Análise do Padrão

A análise do padrão é um pouco mais complexa neste caso. Precisaremos construir as matrizes $\bar{\pi}$ e $\bar{\sigma}$ como na seção 3.3.2, por causa dos dados nas fronteiras. Além dessas matrizes, construímos as matrizes $\bar{\pi}'$ e $\bar{\sigma}'$ que têm informações diferentes das respectivas matrizes da seção 3.3.3.

Por causa da representação fatorada do padrão, será necessária uma etapa de pré-processamento de FP, onde serão obtidas, para cada posição $[i, j]$, $1 \leq i, j \leq m$, os índices j_1 e j_2, $j_1 > j$, $j_2 < j$, $1 < j_1 \leq m + 1$ e $0 \leq j_2 < m$, tais que os sufixos começando em $P[i, j_1]$ e $P[i, j_2]$ estão em ST. Além disso, são obtidos os valores de pos_1 e pos_2 que correspondem às posições de $P[i, j_1]$ e $P[i, j_2]$ na cadeia C. No exemplo 5(e) mostramos os valores calculados para um caso em particular.

Exemplo 5(e) *Matrizes com os valores de j_1, j_2, pos_1 e pos_2 para cada posição $[i, j]$.*

j_1

	1	2	3	4	5	6	7	8
1	3	3	5	5	6	8	8	9
2	3	3	5	5	6	9	9	9
3	3	3	5	5	6	9	9	9
4	2	3	4	5	7	7	9	9
5	2	3	4	5	7	7	9	9
6	2	3	5	5	6	7	9	9
7	2	3	5	5	6	7	9	9
8	2	3	5	5	6	7	9	9

pos_1

	1	2	3	4	5	6	7	8
1	239	239	240	240	241	242	242	243
2	244	244	245	245	246	247	247	247
3	248	248	249	249	250	251	251	251
4	252	253	254	255	256	256	257	257
5	258	259	260	261	262	262	263	263
6	264	265	266	266	267	268	269	269
7	270	271	272	272	273	274	275	275
8	276	277	278	278	279	280	281	281

j_2

	1	2	3	4	5	6	7	8
1	0	0	2	2	4	5	5	7
2	0	0	2	2	4	5	5	5
3	0	0	2	2	4	5	5	5
4	0	1	2	3	4	4	6	6
5	0	1	2	3	4	4	6	6
6	0	1	2	2	4	5	6	6
7	0	1	2	2	4	5	6	6
8	0	1	2	2	4	5	6	6

pos_2

	1	2	3	4	5	6	7	8
1	237	237	238	238	239	240	240	241
2	242	242	243	243	244	245	245	245
3	246	246	247	247	248	249	249	249
4	250	251	252	253	254	254	255	255
5	256	257	258	259	260	260	261	261
6	262	263	264	264	265	266	267	267
7	268	269	270	270	271	272	273	273
8	274	275	276	276	277	278	279	279

Este pré-processamento leva tempo $O(m^2)$, pois apesar de usarmos a representação fatorada FP, precisamos obter os valores para cada posição i, j de P, $1 \leq i, j \leq m$.

No caso seqüencial, construímos uma seqüência $FR = (FP_{l_1}, r_1), \ldots, (FP_{l_\alpha}, r_\alpha)$, onde FP_{l_k} é um bloco que começa na linha l_k com r_k linhas, $1 \leq k \leq \alpha$.

Para o algoritmo CGM, construiremos m seqüências $FR_j = (FP_{l_1,j}, r_{1,j}), \ldots, (FP_{l_{\alpha_j},j}, r_{\alpha_j,j})$, onde FP_{l_k}, j é um sub-bloco de largura $m - j + 1$ que começa na posição l_k, j de P com r_k, j linhas, $1 \leq k \leq \alpha_j$, utilizando o pré-processamento descrito.

Analogamente ao caso seqüencial, teremos as restrições $\hat{FR}_j$ das seqüências FR_j, $1 \leq j \leq m$.

Vamos também obter m seqüências $FR'_j = (FP'_{l_1,j}, r'_{1,j}), \ldots, (FP'_{l_{\alpha'_j},j}, r'_{\alpha'_j,j})$, onde FP'_{l_k}, j é um sub-bloco de largura j que começa na posição $l_k, 1$ de P com r'_k, j linhas, $1 \leq k \leq \alpha'_j$. Para estas seqüências também consideraremos as seqüências $\hat{FR}'_j$. O conteúdo destas seqüências para um caso em particular pode ser acompanhado no exemplo 5(f).

Exemplo 5(f) *Matrizes FR e FR' com valores de início de cada sub-bloco, vetores α e α' com os valores do número de blocos e matrizes r e r' com o numero de repetições de cada linha no bloco.*

FR

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	4
3	4	4	4	4	4	4	4	6
4	6	6	6	6	6	6	6	0

FR'

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2
3	4	4	4	4	4	4	4	4
4	6	6	6	6	6	6	6	6

α

	1	2	3	4	5	6	7	8
α	4	4	4	4	4	4	4	3

α'

	1	2	3	4	5	6	7	8
α'	4	4	4	4	4	4	4	4

r

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	3
2	2	2	2	2	2	2	2	2
3	2	2	2	2	2	2	2	3
4	3	3	3	3	3	3	3	0

r'

	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2
3	2	2	2	2	2	2	2	2
4	3	3	3	3	3	3	3	3

A partir do que foi obtido, poderemos construir as matrizes $\bar{\pi}$, $\bar{\sigma}$, $\bar{\pi}'$ e $\bar{\sigma}'$. O conteúdo destas matrizes, para este algoritmo, difere levemente do conteúdo das mesmas matrizes no algoritmo linear pela forma como trabalhamos as fronteiras em cada caso.

Cada coluna j, $1 \leq j \leq m$, representa:

- na matriz $\bar{\pi}$, o vetor π para a seqüência $\hat{F}R_j$;
- na matriz $\bar{\sigma}$, o vetor σ para a seqüência $\hat{F}R'_j$;
- na matriz $\bar{\pi}'$, o vetor π para a seqüência $\hat{F}R'_j$;
- na matriz $\bar{\sigma}'$, o vetor σ para a seqüência $\hat{F}R_j$.

3.5.2 Análise do Texto

A menos dos procedimentos de busca de padrões com os dados locais e da fronteira, a descrição do algoritmo, neste caso, é praticamente a mesma do algoritmo *Busca_sem_Escala_Bi_CGM* (página 65). O procedimento que combina os dados locais e recebidos do passo 6 é o mesmo.

As listas $Q[j]$ são obtidas para $j \leq n - m + 1$. Porém, as listas $Q[j]$ para os dados contidos nas últimas $m-1$ colunas, isto é, com $n-m+2 \leq j \leq n$, não são obtidas para as colunas j com $j > \lfloor \frac{n}{m} \rfloor m$. Para estas posições, precisamos obter as respectivas listas sem utilizar os vetores B_r e B_l. Os finais de bloco são determinados comparando pares de linhas consecutivas, de uma maneira muito parecida com o que foi feito para se obter os valores de B_r e B_l.

Além destas listas, precisamos construir listas $\bar{Q}[j]$, para $1 \leq j \leq m-1$. Estas listas armazenarão as linhas iniciais de blocos, em ordem decrescente, para cada coluna j. A obtenção destas listas é simétrica à construção das listas $Q[j]$ para as últimas $m-1$ colunas, descrita no parágrafo anterior. Neste caso, as linhas i e $i-1$ são comparadas, para $1 \leq j \leq m-1$ e $2 \leq i \leq n$.

Para determinar as posições candidatas das fronteiras Sul e Sudeste e as posições de confirmação das fronteiras Norte e Nordeste consideramos as sublistas Q_s e Q_n que são determinadas ao se obter as listas $Q[j]$, $1 \leq j \leq n$. Cada lista $Q_s[j]$, para $1 \leq j \leq n$, aponta para o último nó da lista $Q[j]$ cujo valor da primeira linha do bloco não esteja nas fronteiras Sul ou Sudeste. Cada lista $Q_n[j]$, para $1 \leq j \leq n$ aponta para o último nó da lista $Q[j]$ cujo valor da última linha do bloco esteja nas fronteiras Norte ou Nordeste. Para as listas $\bar{Q}[j]$ também obtemos

os apontadores $\bar{Q}_n[j]$ e $\bar{Q}_s[j]$, onde $\bar{Q}_n[j]$ aponta para o último nó da lista $\bar{Q}[j]$ cujo valor da primeira linha do bloco esteja na fronteira Norte, e $\bar{Q}_s[j]$ aponta para o primeiro nó da lista $\bar{Q}[j]$ cujo valor da primeira linha do bloco esteja fora da fronteira Sul.

O exemplo da Figura 5(g) mostra as linhas iniciais e finais dos blocos apontados por $Q_s[j]$, para $1 \leq j \leq \lfloor \frac{n}{m} \rfloor m$. Os exemplos 5(h) e 5(i) mostram estas listas para os dados na fronteira Leste. No exemplo 5(j) temos as listas $\bar{Q}$ para as $m-1$ primeiras colunas.

Exemplo 5(g) *Linhas de início e final de bloco para os nós apontados por* $Q_s[j]$, $1 \leq j \leq \min(n-m+1, \lfloor \frac{n}{m} \rfloor m)$ *(=13).*

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
ini	*12*	*12*	*12*	*12*	*12*	*12*	*12*	*12*	*13*	*12*	*12*	*12*	*12*
fim	*13*	*13*	*13*	*13*	*13*	*13*	*13*	*13*	*13*	*14*	*14*	*14*	*14*

Exemplo 5(h) *Listas para as colunas* j *com* $j > \min(n-m+2, \lfloor \frac{n}{m} \rfloor m)$.

$Q[14] = 1 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 11 \to 12 \to 14 \to 16 \to 17 \to 18 \to 19 \to 20$

$Q[15] = 1 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 11 \to 12 \to 14 \to 16 \to 17 \to 18 \to 19 \to 20$

$Q[16] = 1 \to 3 \to 4 \to 6 \to 7 \to 9 \to 11 \to 14 \to 16 \to 17 \to 18 \to 19 \to 20$

$Q[17] = 1 \to 3 \to 4 \to 6 \to 7 \to 9 \to 11 \to 14 \to 16 \to 17 \to 18 \to 20$

$Q[18] = 1 \to 3 \to 4 \to 6 \to 7 \to 9 \to 11 \to 14 \to 17 \to 18 \to 20$

$Q[19] = 1 \to 3 \to 4 \to 6 \to 7 \to 9 \to 14 \to 17 \to 18 \to 20$

$Q[20] = 3 \to 6 \to 7 \to 9 \to 14 \to 18 \to 20$

Exemplo 5(i) *Linhas de início e final de bloco para os nós apontados por* $Q_s[j]$, $n-m+2 \leq j \leq n$.

	14	*15*	*16*	*17*	*18*	*19*	*20*
ini	*12*	*12*	*10*	*10*	*10*	*8*	*8*
fim	*12*	*12*	*11*	*11*	*11*	*9*	*9*

Exemplo 5(j) *Listas* $\bar{Q}[j]$ *para as* $m-1$ *primeiras colunas.*

$\bar{Q}[1] = 20 \to 16 \to 15 \to 14 \to 11 \to 9 \to 4 \to 1$

$\bar{Q}[2] = 20 \to 16 \to 15 \to 14 \to 12 \to 11 \to 9 \to 7 \to 4 \to 1$

$\bar{Q}[3] = 20 \to 16 \to 15 \to 14 \to 12 \to 11 \to 9 \to 7 \to 5 \to 4 \to 2 \to 1$

$\bar{Q}[4] = 20 \to 16 \to 15 \to 14 \to 12 \to 11 \to 9 \to 7 \to 5 \to 4 \to 2 \to 1$

$\bar{Q}[5] = 20 \to 16 \to 15 \to 14 \to 12 \to 11 \to 9 \to 7 \to 5 \to 4 \to 2 \to 1$

$\bar{Q}[6] = 20 \to 16 \to 15 \to 14 \to 12 \to 11 \to 9 \to 7 \to 5 \to 4 \to 2 \to 1$

$\bar{Q}[7] = 20 \to 18 \to 17 \to 16 \to 15 \to 14 \to\to 1211 \to 9 \to 7 \to 5 \to 4 \to 2 \to 1$

Cada processador obtém, localmente, a lista R das ocorrrências de P em T, utilizando o algoritmo sublinear descrito na seção 3.4. As ocorrências nas regiões de fronteira são obtidas pelas descrições a seguir.

A obtenção dos candidatos a ocorrência, nas regiões de fronteira, difere do caso anterior por causa do formato dos dados de entrada. A seguir, descreveremos os procedimentos para determinação dos candidatos nas fronteiras Sul e Leste.

PROCEDIMENTO: Fronteira2_S
Entrada: a cadeia C e a seqüência $\hat{F}R$.
Saída: uma lista C_S das posições candidatas a ocorrência de $\hat{F}R$ em FS na fronteira Sul.
{ $\hat{F}R = (FP_{l_2}, r_2) \ldots (FP_{l_{\alpha-1}}, r_{\alpha-1})$. }
1. **para** $j \leftarrow 1$ **até** $n - m + 1$ **faça**
2. $\quad linha[0] \leftarrow Q_s[j].linha$
3. $\quad q \leftarrow Próximo(Q_s[j])$
4. $\quad i \leftarrow q.linha + 1$
5. $\quad k \leftarrow 1$
6. $\quad cont \leftarrow 1$
7. $\quad linha[cont] \leftarrow q.linha$
8. $\quad$ **enquanto** $i < n$ **e** $Próximo(q).linha < n$ **faça**
9. $\quad\quad$ **enquanto** $k > 1$ **e** $\neg Compara(i, k+1, j, Próximo(q), r_{k+1})$ **faça**
10. $\quad\quad\quad k \leftarrow \bar{\pi}[k][1]$
11. $\quad\quad$ **se** $Compara(i, k+1, j, Próximo(q), r_{k+1,1})$
12. $\quad\quad\quad k \leftarrow k + 1$
13. $\quad\quad cont \leftarrow cont + 1$
14. $\quad\quad q \leftarrow Próximo(q)$
15. $\quad\quad i \leftarrow q.linha + 1$
16. $\quad\quad linha[cont] \leftarrow q.linha$
17. **enquanto** $k > 1$ **faça**
18. $\quad Insere((linha[cont - k + 1] - r_{1,1} + 1, j), C_S)$
19. $\quad k \leftarrow \bar{\pi}[k][1]$
20. **devolva** C_S
fim procedimento

O procedimento para obtenção dos candidatos da fronteira Sul percorre as colunas j entre 1 e $n - m + 1$ e as linhas armazenadas nas listas $Q[j]$ a partir do nó apontado por $Q_s[j]$. A lista $Q[j]$ é percorrida até que $i \geq n$ ou $Próximo(q).linha \geq n$, com isto o último bloco na coluna j não é comparado com nenhum bloco de $\hat{F}R$. A partir daí, o laço **enquanto** da linha 17 insere os candidatos a ocorrência de $\hat{F}R$ em FS na posição $[linha[cont - k + 1] + 1, j]$.

Precisamos, agora, testar se os elementos da lista C_S são candidatos a ocorrência de P em S na fronteira Sul. Seja β o número de blocos da ocorrência em C_S mais 2, $\beta < \alpha_{1,1}$. Estas duas unidades a mais são por causa do primeiro bloco da ocorrência de P em S e do último bloco do texto, que não foram contados na obtenção desta ocorrência em C_S. Então, comparamos o bloco antes da ocorrência com o bloco $(FP_{l_{1,1}}, r_{1,1})$, comparamos a primeira linha do último bloco do texto com a linha $l_{\beta,1}$, e verificamos se o número de linhas deste último bloco é menor ou igual a $r_{\beta,1}$. Se o resultado destas verificações for afirmativo, o elemento é atualizado em C_S; caso contrário, ele é removido.

Por causa do último bloco do texto na coluna j não estar envolvido no procedimento e estarmos procurando uma ocorrência de $\hat{FR}$, precisamos ainda verificar se não existe uma ocorrência de FP a partir do penúltimo bloco, ou a partir do último bloco. Isto é feito verificando-se, no primeiro caso, se existe uma ocorrência de $(FP_{l_1,1}, r_{1,1})$ no penúltimo bloco, se a primeira linha do último bloco do texto e a segunda linha de $\hat{FR}$ são iguais e se o número de linhas deste último bloco é menor ou igual a $r_{2,1}$. No segundo caso, basta verificar se a primeira linha do último bloco do texto e a primeira linha de $\hat{FR}$ são iguais. As posições de ocorrência são incluídas na lista C_S.

PROCEDIMENTO: Fronteira2_E
Entrada: a cadeia C e a seqüência $\hat{FR}$.
Saída: a lista C_E das posições candidatas a ocorrência de $\hat{FR}$ em FS na fronteira Leste.
$\{\ \hat{FR} = (FP_{l_2}, r_2) \ldots (FP_{l_{\alpha-1}}, r_{\alpha-1}).\ \}$

1. **para** $j \leftarrow n - m + 2$ **até** n **faça**
2. $\quad linha[0] \leftarrow 0$
3. $\quad q \leftarrow Q[j]$
4. $\quad i \leftarrow q.linha + 1$
5. $\quad k \leftarrow 1$
6. $\quad cont \leftarrow 1$
7. $\quad linha[cont] \leftarrow q.linha$
8. $\quad$ **enquanto** $i < n$ **e** $Próximo(q) \neq fim$ **faça**
9. $\quad\quad$ **enquanto** $k > 1$ **e** $\neg Compara(i, k+1, j, Próximo(q), r'_{k+1,n-j+1})$ **faça**
10. $\quad\quad\quad k \leftarrow \bar{\pi}[k][j - n + m]$
11. $\quad\quad$ **se** $Compara(i, k+1, j, Próximo(q), r'_{k+1,n-j+1})$
12. $\quad\quad\quad k \leftarrow k + 1$
13. $\quad\quad cont \leftarrow cont + 1$
14. $\quad\quad q \leftarrow Próximo(q)$
15. $\quad\quad i \leftarrow q.linha + 1$
16. $\quad\quad linha[cont] \leftarrow q.linha$
17 $\quad\quad$ **se** $k = \alpha'_{n-j+1} - 1$
18. $\quad\quad\quad Insere((linha[cont - k + 1] - r'_{1,n-j+1} + 1, j), R)$
19. $\quad\quad\quad k \leftarrow \bar{\pi}[k][j - n + m]$
20. **devolva** C_E

fim procedimento

Os candidatos da fronteira Leste são obtidos percorrendo-se as colunas j, com $n - m + 2 \leq j \leq n$ e as listas $Q[j]$ para estas colunas. O funcionamento do procedimento é análogo ao funcionamento do procedimento *Busca_*$\hat{FR}$ (página 84), com a diferença que para cada coluna j, buscamos uma ocorrência de $\hat{FR}'_{j-n+m}$ em $S[1, \ldots, n; j]$. A determinação das ocorrências de FP na fronteira Leste é obtida comparando-se os blocos de texto antes e depois da ocorrência com $(FP'_{l_1,j-n+m}, r_{1,j-n+m})$ e $(FP'_{l'_\alpha,j-n+m}, r'_{\alpha',j-n+m})$. Um resultado afirmativo em ambas as comparações atualiza a posição na lista C_E; caso contrário, ela é removida.

Entre as posições candidatas a serem determinadas faltam ainda aquelas dentro da fronteira Sudeste. Estas posições são determinadas pelo procedimento *Fronteira2_SSEE* que também determina as posições candidatas nas fronteiras Sul e Leste.

A descrição deste procedimento é formada pelas descrições dos procedimentos *Fronteira2_S* e *Fronteira2_E* e pela inclusão de um laço que verifica as ocorrências na fronteira Sudeste. Estas posições são verificadas, logo após o correspondente laço **enquanto** da linha 8 do procedimento *Fronteira2_E*, através do laço descrito a seguir.

$\vdots$
enquanto $k > 1$ **faça**
  $Insere((linha[cont - k + 1] - r'_{1,n-j+1} + 1, j), C_{SE})$
  $k \leftarrow \bar{\pi}[k][j - n + m]$
$\vdots$

Para se determinar as posições candidatas a ocorrências de P em S na fronteira Sudeste, verificam-se as extensões como no caso da fronteira Sul. Além disso, deve-se verificar a existência de candidatos nos penúltimos e últimos blocos de cada coluna da fronteira Sudeste.

As posições de confirmação serão determinadas pelos procedimentos *Fronteira2_N*, *Fronteira2_O* e *Fronteira2_ONON*. Nas fronteiras Norte e Nordeste utilizaremos as listas $Q_n[j]$, para $1 \leq j \leq n$. Para as fronteiras Oeste, Sudoeste e Noroeste utilizaremos as listas $\bar{Q}[j]$, $\bar{Q}_s[j]$ e $\bar{Q}_n[j]$, respectivamente, para $1 \leq j \leq m - 1$. Serão também necessárias as matrizes $\bar{\sigma}$, $\bar{\pi}'$ e $\bar{\sigma}'$. As posições de confirmação estarão posicionadas, em relação às posições candidatas, como na seção 3.3.3 (página 67).

O funcionamento do procedimento *Fronteira2_N* é similar ao funcionamento do procedimento *Fronteira2_S*. Neste procedimento, para $1 \leq j \leq n$ percorremos a lista $Q[j]$ a partir do elemento apontado por $Q_n[j]$ em direção ao início da lista de forma a determinar se existe uma posição de confirmação para ocorrências de $\hat{FR}$ em S nas fronteiras Norte e Nordeste, utilizando a matriz $\bar{\sigma}'$ ao invés da matriz $\bar{\pi}$. Cada posição de confirmação é armazenada na lista C_N ou C_{NE}. Notemos que, neste procedimento, o primeiro bloco de S em cada coluna não é comparado com nenhum bloco de $\hat{FR}$.

As posições de confirmação de FP em S nestas fronteiras, para cada elemento de C_N ou C_{NE}, são determinadas como descrito a seguir. Para cada posição i, j armazenada em C_N ou C_{NE}, seja β a posição em $Q[j]$, a partir do início desta, do nó q que contém i. A posição $[i, j]$ terá seu valor atualizado na lista C_N ou C_{NE} se as seguintes condições ocorrerem: qualquer linha do bloco seguinte ao apontado por q for igual a $FP_{l_{\alpha},x}$, a altura deste bloco for maior ou igual a $r_{\alpha,x}$, qualquer linha do primeiro bloco for igual a $FP_{l_{\alpha-\beta-1},x}$ e a altura do primeiro bloco for menor ou igual a $r_{\alpha-\beta-1,x}$, onde $x = 1$ se a posição $[i, j]$ estiver na fronteira Norte e $x = n - j + 1$ se a posição $[i, j]$ estiver na fronteira Nordeste. Se qualquer uma das verificações falhar, a posição é removida da respectiva lista.

Como o primeiro bloco do texto, em qualquer coluna j, não é comparado com nenhum bloco de $\hat{FR}$ e estamos procurando ocorrências de $\hat{FR}$, não obteremos posições de confirmação que

estejam no primeiro ou no segundo bloco de texto. Para verificar se existe uma posição de confirmação na coluna j do segundo bloco verificamos se qualquer linha deste bloco é igual a linha $FP_{l_{\alpha},x}$, se a altura deste bloco é maior ou igual a $r_{\alpha,x}$, se qualquer linha do primeiro bloco é igual a $FP_{l_{\alpha-1},x}$ e se a altura deste bloco é menor ou igual a $r_{\alpha-1,x}$. No caso de existir uma posição de confirmação na coluna j do primeiro bloco, ela será determinada ao verificarmos se qualquer linha do primeiro bloco é igual à linha $FP_{l_{\alpha},x}$ e se a altura deste bloco é maior ou igual $r_{\alpha,x}$. Em ambos os casos, o valor de x depende de onde a posição de confirmação está sendo determinada: na fronteira Norte ou na Nordeste.

A determinação das posições de confirmação do procedimento *Fronteira2_O* tem comportamento similar ao procedimento *Fronteira_E* (página 66). Neste caso, vamos determinar as posições de confirmação presentes nas fronteiras Oeste e Sudoeste que servirão para certificar as posições candidatas nas fronteiras Leste e Sudeste do vizinho Oeste.

O procedimento utiliza as listas $\bar{Q}[j]$ e $\bar{Q}_s[j]$, $1 \leq j \leq m-1$, para percorrer os blocos destas colunas. Para a determinação das posições de ocorrência de $\hat{FR}$ em $S[1,\ldots,n;j]$, percorremos as listas $\bar{Q}[j]$, em ordem decrescente dos valores das representantes, visitando cada sub-bloco, e comparando-os com os sub-blocos de $\hat{FR}_j$ de maneira análoga, e restrita aos sub-blocos, ao que é feito no procedimento *Busca_*$\hat{FR}$ (página 84). Ao final, teremos as posições de ocorrências armazenadas na lista C_O.

Cada posição $[i,j]$ deve ser, então, verificada se pode ser estendida a uma ocorrência de FR_j. Isto é feito comparando as representantes dos sub-blocos antes e depois da ocorrência com as linhas $l_{1,j}$ e $l_{\alpha_j,j}$, respectivamente, e se as alturas destes sub-blocos são maiores ou iguais a $r_{1,j}$ e $r_{\alpha_j,j}$, respectivamente. Em caso afirmativo, estas posições são atualizadas em C_O e caso contrário, são dela removidas.

Os procedimentos vistos até agora não abrangem a fronteira Noroeste. No procedimento *Fronteira2_ONON*, estas posições são obtidas, além das posições de confirmação das demais fronteiras.

Estas posições são obtidas de uma maneira simétrica à fronteira Nordeste. Neste caso, utilizamos as listas apontadas por $\bar{Q}_n[j]$ e as percorremos em ordem decrescente das representantes dos blocos, para determinar posições de confirmação que são armazenadas em C_{NO}. Estas posições são ocorrências de $\hat{FR}_j$ na coluna j da fronteira Noroeste. As extensões a ocorrências de FR_j são obtidas de modo similar e atualizadas ou removidas da lista C_{NO}. Novamente, o primeiro sub-bloco não é comparado com sub-blocos de $\bar{Q}_n[j]$ e as posições de confirmação que, eventualmente, estejam presentes no primeiro ou segundo sub-blocos do texto serão obtidas separadamente.

Depois de obtidas as posições de confirmação em todas as fronteiras necessárias, as listas são enviadas para os vizinhos conforme os índices de cada processador. Este passo é o mesmo passo 5 do algoritmo *Busca_sem_Escala_Bi_CGM* (página 65). De posse das listas localmente calculadas e das listas recebidas, aplica-se o procedimento *Combina_Bi* (página 71) como na seção 3.3.3 e teremos em cada processador as posições de ocorrência de P em S.

3.5.3 Tempo e Corretude

Como o tempo e a corretude do algoritmo desta seção dependem de outros algoritmos já vistos, exigindo algumas alterações e extensões, as demonstrações estão aqui expostas de uma maneira semi-formal.

A corretude do algoritmo, como na seção 3.3, depende da corretude dos procedimentos envolvidos em sua descrição. As ocorrências locais não pertencentes às regiões de fronteira são corretamente obtidas pelo procedimento *Busca_$\hat{F}R$* (página 84). O procedimento *Combina_Bi* é o mesmo da seção 3.3.3 e sua corretude já foi vista. Resta verificar a corretude dos procedimentos que obtêm as posições candidatas e de confirmação nas regiões de fronteira.

Vamos analisar o que ocorre na obtenção das posições candidatas na fronteira Sul. O procedimento *Fronteira2_S* (página 91) obtém as posições de ocorrência de $\hat{F}R_1$ em S. Temos a garantia de que todos os blocos estão corretamente listados em $Q[j]$, $1 \leq j \leq n - m + 1$. Além disso, $Q_s[j]$ aponta pra o último bloco de $Q[j]$ que não está inteiramente contido na fronteira Sul e que não pode ser uma ocorrência de $\hat{F}R_1$. O laço **enquanto** da linha 8 do procedimento tem o mesmo funcionamento do respectivo laço do procedimento *Busca_$\hat{F}R$* (página 84), a menos do teste de parada do laço. Por argumentos análogos à corretude do procedimento *Busca_$\hat{F}R$*, o procedimento *Fronteira2_S* (página 91) tem sua corretude baseada na corretude do procedimento *Fronteira_S* (página 66). Assim, as posições armazenadas em C_S são posições de ocorrência de $\hat{F}R_1$ em S, corretamente obtidas pelo procedimento.

Para verificar se estas posições são candidatas a ocorrência de FR_1 em S na fronteira Sul, os blocos antes e depois da ocorrência em C_S são simplesmente comparados com os respectivos blocos de FR_1 e a posição é corretamente atualizada se e somente se os resultados destas comparações são verdadeiros.

As ocorrências de posições candidatas nos penúltimos e últimos blocos de texto são obtidas pela simples comparação com os dois primeiros blocos de FR_1 e são corretamente obtidos.

Com isto, temos que qualquer posição na fronteira Sul que seja candidata a ocorrência está armazenada em C_S. Nenhuma posição é incorretamente inserida, pois o procedimento *Fronteira2_S* (página 91) somente insere aquelas posições que são ocorrências de $\hat{F}R_1$ em S e no caso das ococrrências nos dois últimos blocos de S, eles só são inseridos se realmente forem candidatos a ocorrência. Nenhuma posição é incorretamente atualizada pois são feitas comparações entre blocos e somente aquelas posições que as satisfizerem serão atualizadas.

As corretudes dos demais procedimentos serão omitidas pois têm sua justificativa obtida de maneira similar.

O tempo local e a memória utilizada em cada processador para se obter todas as ocorrências de P em T é descrita a seguir.

O pré-processamento da entrada, que envolve a construção da árvore de sufixos, pré-processamento para consultas LCA, obtenção dos vetores B_r e B_l e suas respectivas árvores Cartesianas leva tempo $O(|C_b| + \lfloor \frac{n^2}{m} \rfloor + |FS|)$, onde C_b é definido como no caso seqüencial para os dados locais. A árvore de sufixos, já pré-processada para consultas LCA, ocupa espaço $O(|C_b|)$. B_r e

B_l ocupam espaço no máximo $O(n^2)$.

A análise do padrão, para este caso, envolve a obtenção das matrizes j_1, j_2, pos_1 e pos_2 para a matriz padrão, o que leva tempo $O(m^2)$. Estas matrizes são armazenadas em $O(m^2)$ posições. A obtenção das seqüências FR_j e FR'_j leva tempo $O(|FR_j|)$ e $O(|FR'_j|)$, respectivamente. Como são m seqüências, o tempo total para este passo é $O(m^2)$, ocupando memória $O(m^2)$. As matrizes π, $\bar{\pi}$, σ e $\bar{\sigma}$ são obtidas em tempo $O(m^2)$ e ocupam espaço $O(m^2)$.

A obtenção das listas $Q[j]$, para cada coluna j, $1 \leq j \leq n-m+1$, leva tempo $O(|Q[j]|)$, e para obter todas estas listas leva-se tempo $O(\sum_{j=1}^{n-m+1} |Q[j]|)$. As listas $Q[j]$, para $n-m+2 \leq j \leq n$, e $\bar{Q}[j]$, para $1 \leq j \leq m-1$, são obtidas em tempo no máximo $O(nm)$. Os apontadores $Q_s[j]$, $Q_n[j]$ e $\bar{Q}_s[j]$, $\bar{Q}_n[j]$ são obtidos durante as construções das listas $Q[j]$, para $1 \leq j \leq n$, e $\bar{Q}[j]$, para $1 \leq j \leq m-1$, respectivamente. Logo, estas listas são obtidas em tempo $O(nm+\sum_{j=1}^{n-m+1} |Q[j]|)$ e ocupam espaço no máximo $O(n^2)$, pois $\sum_{j=1}^{n-m+1} |Q[j]| \leq n^2$.

O pré-processamento adicional leva tempo $O(|FS| + nm + \sum_{j=1}^{n-m+1} |Q[j]|)$, consumindo memória $O(\sum_{j=1}^{n-m+1} |Q[j]|)$.

O procedimento para obtenção das ocorrências nas regiões fora das fronteiras leva tempo $O(\sum_{j=1}^{n-m+1} |Q[j]|)$, gastando memória $O(|R|) \leq O(n^2)$. O procedimento *Fronteira2_S* leva tempo $O(\sum_{j=1}^{n-m+1} |Q_s[j]|)$, gastando memória $O(|C_S|) \leq O(nm)$. O procedimento *Fronteira2_E* é executado em tempo $O(\sum_{j=n-m+2}^{n} |Q[j]|)$, utilizando $O(|C_E|) \leq O(nm)$ de memória. O procedimento *Fronteira2_SSEE* gasta tempo $O(\sum_{j=1}^{n-m+1} |Q[j]|) + O(\sum_{j=n-m+2}^{n} |Q[j]|) + O(\sum_{j=n-m+2}^{n} |Q_s[j]|)$, para obter as posições candidatas nas fronteiras Sul, Leste e Sudeste, utilizando memória $O(|C_S| + |C_E| + |C_{SE}|) \leq O(nm)$.

As posições de confirmação nas fronteiras Norte e Nordeste são obtidas pelo procedimento *Fronteira2_N* em tempo $O(\sum_{j=1}^{n} |Q_n[j]|)$ consumindo memória $O(|C_N| + |C_{NE}|) \leq O(nm)$. O procedimento *Fronteira2_O* obtém as posições de confirmação nas fronteiras Oeste e Sudoeste em tempo $O(\sum_{j=1}^{m-1} |\bar{Q}[j]|) + O(\sum_{j=1}^{m-1} |\bar{Q}_s[j]|)$ e consome $O(|C_O| + |C_{SO}|) \leq O(nm)$ de memória. Finalmente, o procedimento *Fronteira2_ONON* obtém as posições candidatas nas fronteiras Norte, Nordeste, Oeste, Sudoeste e Noroeste em tempo $O(\sum_{j=1}^{n} |Q_n[j]|) + O(\sum_{j=1}^{m-1} |\bar{Q}[j]|) + O(\sum_{j=1}^{m-1} |\bar{Q}_s[j]|) + O(\sum_{j=1}^{m-1} |\bar{Q}_n[j]|)$, consumindo memória $O(|C_N| + |C_{NE}| + |C_O| + |C_{SO}| + |C_{NO}|) \leq O(nm)$.

O procedimento *Combina_Bi* é executado em tempo $O(nm)$ utilizando $O(n^2)$ de memória.

O algoritmo utiliza uma rodada de comunicação, onde são enviadas as posições de confirmação localmente obtidas para os respectivos vizinhos, e que envolve no máximo $O(nm)(\leq O(n^2))$ dados.

Portanto, todas as ocorrências de P em T podem ser obtidas pelo algoritmo, que é executado em tempo local $O(n^2) = O(N^2/p)$, utilizando no máximo memória $O(n^2) = O(N^2/p)$ e uma rodada de comunicação onde são enviados no máximo $O(nm) \leq O(N^2/p)$ valores.

3.5.4 Resultados da Implementação

Para os testes da implementação da versão CGM para o algoritmo seqüencial sublinear, utilizamos uma matriz texto de ordem $N = 96$ e matrizes padrão de ordem $m = 4$ e $m = 24$. As matrizes foram constituídas pelos símbolos a e b. As linhas das matrizes foram formadas por estes símbolos alternadamente. Duas linhas consecutivas não começavam com o mesmo símbolo e qualquer uma das matrizes tinha o símbolo a na primeira linha e primeira coluna. Com estas matrizes, o número de ocorrências é $N/2 - m$ para cada uma das $N - m$ primeiras linhas.

Nos testes, obtivemos os tempos absolutos em segundos expressos no gráfico da Figura 3.8 e cujas tabelas de valores estão no apêndice A. No gráfico, podemos observar que o comportamento é similar ao que ocorre na versão CGM do algoritmo seqüencial linear. A execução seqüencial apresenta tempo maior para um padrão de ordem $m = 4$, uma vez que a quantidade de computação local é maior para este caso e não ocorrem comunicações. Para 4 e 16 processadores, por causa da comunicação, o tempo para $m = 4$ se torna menor. Observamos, também, que a queda no tempo é maior para o padrão de ordem $m = 4$, quando utiliza-se 4 processadores. Tanto para $m = 4$ quanto para $m = 24$ a queda no tempo é mais sensível quando passamos de 1 para 4 processadores.

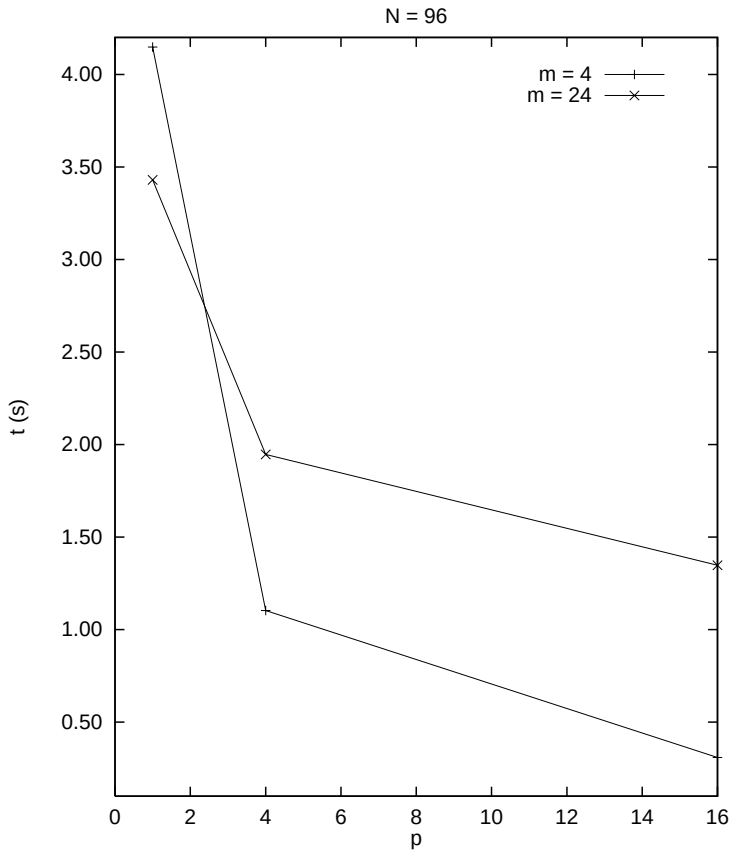

Figura 3.8: Tempo em s para $N = 96$, $m = 4$ e $m = 24$, utilizando 1, 4 e 16 processadores.

O *speedup* está mostrado no gráfico da Figura 3.9. Novamente temos um *speedup* muito próximo ao ótimo para 4 processadores quando a ordem do padrão é pequena ($m = 4$). Este *speedup* é bem mais discreto para $m = 24$, uma vez que o volume de dados comunicados neste exemplo é grande.

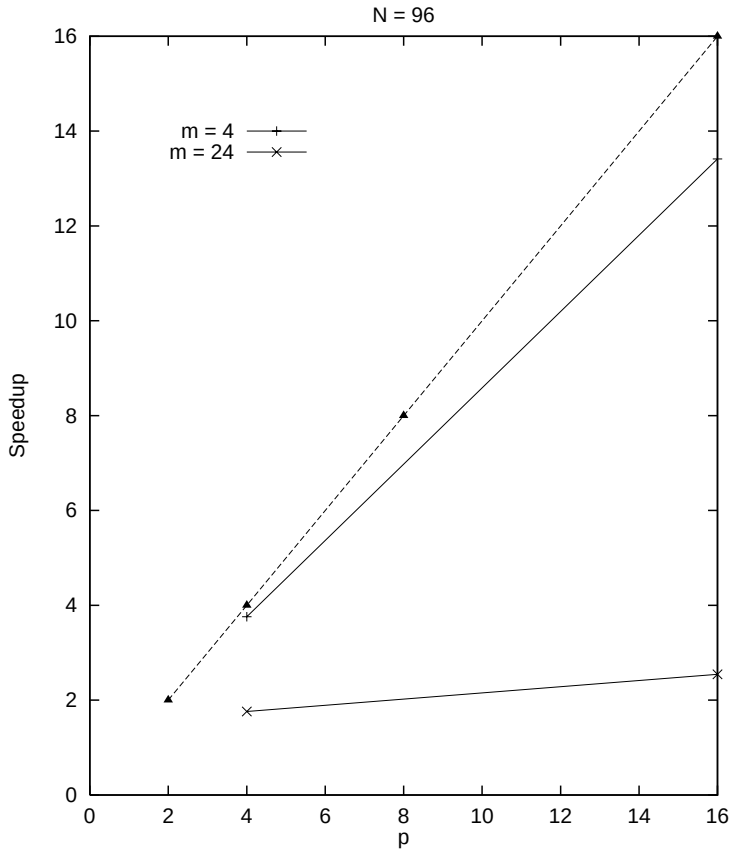

Figura 3.9: *Speedup* para 4 e 16 processadores com $N = 96$, $m = 4$ e $m = 24$.

3.6 Notas

Neste capítulo, apresentamos dois algoritmos seqüenciais para o problema de busca bidimensional de padrões sem escala: um de tempo linear e outro de tempo sublinear (sob algumas condições nos dados de entrada). O segundo algoritmo é importante para a descrição do algoritmo para o caso bidimensional com escala, apresentado no próximo capítulo. Assim , adiantamos alguns conceitos e idéias que serão utilizados no caso com escala e que são mais simples de serem vistos no caso sem escala inicialmente.

Para ambos os algoritmos, construímos versões CGM que rodam em tempo $O(n^2 + m^2) = O(N^2/p + m^2)$, consumindo memória $O(n^2 + m^2) = O(N^2/p + m^2)$ e utilizando uma rodada de

comunicação, onde são trocados no máximo $O(nm) = O(N^2/\sqrt{p}m)$ dados.

Utilizando a idéia contida nas notas do capítulo anterior (seção 2.6), podemos modificar os algoritmos, ou mesmo combiná-los, de forma a reduzir a quantidade de dados comunicados.

A implementação dos algoritmos CGM revelou um *speedup* significativo em vista da complexidade maior dos algorimtos seqüenciais, principalmente por causa da construção da árvore de sufixos.

Capítulo 4

Busca Bidimensional de Padrões com Escala

4.1 Introdução

O problema de busca bidimensional de padrões com escala tem como entrada uma matriz padrão $P_{i,j}$, com $1 \leq i, j \leq m$, e uma matriz texto $T_{i,j}$, com $1 \leq i, j \leq N$, onde $m \leq N$, e tem como saída todas as posições de T onde existem k-ocorrências de P, para todo $k = 1, \ldots \lfloor N/m \rfloor$.

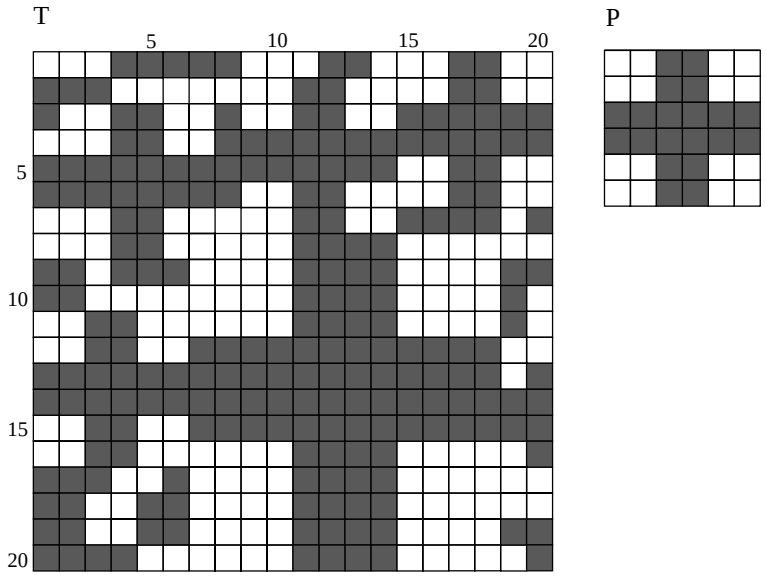

Figura 4.1: Exemplo de figuras armazenadas nas matrizes texto T e padrão P com ocorrências de P em T nas posições $[1, 15]$, $[2, 9]$, $[3, 2]$, $[11, 1]$ com escala 1 e na posição $[8, 7]$ com escala 2.

Na seção 4.2, apresentamos um algoritmo seqüencial de complexidade $O(N^2)$, proposto por

Amir *et al* [8], que resolve o problema de busca bidimensional de padrões com escala. Este algoritmo é uma generalização do algoritmo visto na seção 3.4. A linearidade do algoritmo é obtida ao reduzir-se, para cada escala k, o número de colunas nas quais pode existir o início de uma k-ocorrência e dentro destas colunas reduz-se a quantidade de posições onde pode existir uma k-ocorrência.

Diferentemente do caso unidimensional, as escalas de cada ocorrência não são determinadas durante a execução. As k-ocorrências são determinadas separadamente para cada escala k.

Com este algoritmo seqüencial construímos, na seção 4.3, um algoritmo CGM para o problema, que mantém a linearidade no processamento local e utiliza uma rodada de comunicação. Este algoritmo constitui-se na contribuição mais importante desta tese. A descrição deste algoritmo é baseada no algoritmo descrito na seção 3.5 e adaptado para lidar com as escalas.

O algoritmo CGM foi implementado na máquina *Parsytec PowerXplorer* e os resultados obtidos estão expostos na seção 4.4, onde podemos ver um *speedup* bastante significativo.

Ao final do capítulo tecemos algumas considerações gerais.

4.2 Algoritmo Seqüencial

Antes de detalhar as principais idéias do algoritmo, proposto por Amir *et al* [8], precisamos introduzir conceitos novos, alguns deles derivados dos conceitos utilizados no algoritmo seqüencial sublinear para busca bidimensional de padrões sem escala. Inicialmente, vamos generalizar o conceito de blocos, de forma a contemplar as escalas.

Definição 11 (k-bloco) *Seja k um inteiro positivo. Um* **k-*bloco*** *na posição i_1 de uma coluna j_1 de T é a submatriz $T' = T[i_1, \ldots, i_2; j_1, \ldots, j_1 + km - 1]$ que satisfaz as seguintes condições:*

1. *todas as linhas de T' são iguais;*
2. *nenhuma linha pode ser adicionada a T' sem corromper a condição 1, isto é, $T[i_2; j_1, \ldots, j_1 + km-1] \neq T[i_2+1; j_1, \ldots, j_1+km-1]$ e $T[i_1; j_1, \ldots, j_1+km-1] \neq T[i_1-1; j_1, \ldots j_1+km-1]$.*

A *altura* de uma submatriz é o seu número de linhas. A altura de tais k-blocos é $i_2 - i_1 + 1$. Se um k-bloco está contido em uma k-ocorrência então sua altura deve ser no mínimo k, porque cada linha do padrão deve aparecer k vezes sucessivas.

O algoritmo tem como entrada a representação fatorada FT da matriz texto T e a representação fatorada FP da matriz padrão P. As principais etapas do algoritmo são:

1. Construção das estruturas de dados (cadeia C e árvore de sufixos).
2. Análise do Padrão.
3. Análise do Texto.

Nas subseções a seguir descreveremos com mais detalhes cada etapa.

4.2.1 Construção das Estruturas de Dados

A partir dos dados de entrada, constrói-se uma cadeia C obtida pela concatenação de todas as linhas de FT seguidas pelas $\lfloor N/m \rfloor$ seguintes cadeias:

1. uma concatenação das linhas de FP.
2. para cada k, $2 \leq k \leq \lfloor N/m \rfloor$, uma concatenação das linhas de P , onde cada linha é escalada a k, e dada em sua representação fatorada.

Constrói-se a árvore de sufixos ST de C e a ela aplica-se o algoritmo de LCA. Isto é feito para permitir tempo constante na comparação entre uma subcadeia do texto e uma linha de P^k, $1 \leq k \leq \lfloor N/m \rfloor$.

Construímos $B_r[i,c]$ e $B_l[i,c]$, para $i = 1, \ldots, N$ e $c = m, 2m, \ldots, \lfloor N/m \rfloor$, como no algoritmo da seção 3.4.

No algoritmo seqüencial sublinear para busca bidimensional sem escala são utilizadas árvores Cartesianas construídas para estes vetores B_r e B_l, para cada coluna potência c, com o objetivo de guiar a divisão em blocos. Para o caso com escala, em que devemos dividir em k-blocos, a árvore Cartesiana contém mais informações do que é necessário. Como isto é muito custoso, a definição de árvore Cartesiana é modificada de modo que blocos com altura menor do que k não são considerados.

Definição 12 (Árvore k-Altura Cartesiana) *Uma* ***árvore k-altura Cartesiana*** *de uma lista $L = l_1, \ldots, l_N$ é uma árvore binária com raiz definida recursivamente como se segue:*

Seja $l_{raiz} = \min\{l_1, \ldots, l_N\}$ (se $\min\{l_1, \ldots, l_N\}$ é obtido em mais de uma posição, então a raiz é aquela com maior índice). Então,

- *l_{raiz} é a raiz da árvore k-altura Cartesiana*
- *o filho esquerdo da raiz é a árvore k-altura Cartesiana de $[l_1, \ldots, l_{raiz-k}]$*
- *o filho direito da raiz é a árvore k-altura Cartesiana de $[l_{raiz+k}, \ldots, l_N]$*

Como veremos na seção 4.2.5, teremos várias listas para as quais serão construídas árvores k-altura Cartesianas. Por razões de eficiência serão geradas tantas árvores k-altura quanto necessárias, durante a análise do texto, de forma que o número de operações seja proporcional ao número de nós que são realmente necessários em cada árvore k-altura. Para isto, são necessárias consultas do mínimo intervalar em tempo constante. Assim, pré-processamos os vetores

$B_r[1,\ldots,N;c]$ e $B_l[1,\ldots,N;c]$, para cada coluna potência c, utilizando o algoritmo descrito na seção 1.8.

Além disso, nesta etapa de construção das estruturas de dados, precisamos de mais algumas estruturas cuja utilidade será vista mais adiante. A representação fatorada nos permite construir, para cada coluna potência c, os seguintes vetores:

1. $D_r[i,c]$ é igual ao número de repetições sucessivas do símbolo em $T[i,c]$, iniciando na posição $[i,c]$ e para a direita. Isto é, $T[i,c] = T[i,c+1] = \cdots = T[i+D_r[i,c]-1] \neq T[i,c+D_r[i,c]]$.
2. $D_l[i,c]$ é igual ao número de repetições do símbolo em $T[i,c]$, iniciando em $[i,c]$ e para a esquerda.

Para cada coluna potência c, os vetores $D_r[1,\ldots N;c]$ e $D_l[1,\ldots,N;c]$ são também pré-processados para consultas de mínimo intervalar.

4.2.2 Análise do Padrão

Esta etapa é a mesma da seção 3.4.2. A escala não afeta a construção do vetor π, mas, como veremos, a comparação entre blocos do padrão e blocos do texto é diferente, pela escala estar envolvida.

4.2.3 Análise do Texto

Esta etapa é dividida em dois casos, conforme os dados do padrão. Vamos supor que existam pelo menos duas linhas, i_1 e i_2, do padrão, onde $P[i_1;1,\ldots,m] \neq P[i_1+1;1,\ldots,m]$ e $P[i_2;1,\ldots,m] \neq P[i_2+1;1,\ldots,m]$. Caso esta suposição não seja válida, a solução é facilmente obtida. Além disso, esta solução fácil nos permite supor que ao menos uma das linhas do padrão tem comprimento fatorado maior do que um, isto é, existe pelo menos uma linha cujos símbolos não sejam todos iguais. Se cada linha do padrão é formada pela repetição de um símbolo, então podemos ver o padrão rotacionado de 90° como um padrão sem mudança de linhas. A solução fácil pode então ser aplicada a este caso. Dentro da suposição, consideramos dois casos:

Caso A. Em quaisquer $m/2$ linhas sucessivas do padrão existe pelo menos uma linha de comprimento fatorado maior do que 1.

Caso B. Caso contrário. Isto é, existem pelo menos $m/2$ linhas consecutivas do padrão com comprimento fatorado 1.

A idéia do algoritmo segue alguns conceitos vistos no algoritmo seqüencial sublinear para buscas bidimensionais sem escala (seção 3.4). Diferentemente do caso unidimensional, o algorimo considera cada escala k, $1 \leq k \leq \lfloor N/m \rfloor$, separadamente. Nosso objetivo inicial, antes de buscar as k-ocorrências, é reduzir o número de posições onde estas possam existir.

Para o caso A, o funcionamento do algoritmo, para cada escala k, é basicamente o mesmo do algoritmo sublinear. As diferenças residem, justamente, na presença da escala. Inicialmente, reduzimos, no texto, as "regiões" de busca do padrão com escala. No algoritmo sublinear esta redução constava da determinação, para todas as colunas, de listas de finais de blocos. Aqui, temos a escala a considerar. Para $k = 1$, todas as colunas são selecionadas. Para os outros valores de k, apenas $\lfloor N/k \rfloor$ colunas são selecionadas. Nestas colunas, determinamos listas de pares de índices de linhas, denominados *intervalos de linha*, que podem conter finais de k-blocos. A partir destas listas de intervalos, fazemos uma segunda redução onde obtemos listas de finais de k-blocos, utilizando árvores k-altura Cartesianas. O algoritmo, então, utiliza estas listas, como no algoritmo sublinear, para obter as k-ocorrências. O objetivo destas reduções é, ao final, obter o tempo linear para a determinação de todas as k-ocorrências, como veremos no cálculo do tempo do algoritmo (seção 4.2.8).

No caso B, o padrão é mais simples. Utilizamos, também, uma redução nas "regiões" de busca. Nesta redução, utilizamos o algoritmo de busca unidimensional de padrões com escala para a exclusão de posições onde não podem haver k-ocorrências do padrão, diminuindo assim, a quantidade de posições a serem testadas como k-ocorrências e levando a um tempo linear na determinação das k-ocorrências.

Observemos que, em ambos os casos, a idéia para se obter o tempo linear é, antes de buscar as k-ocorrências, diminuir a quantidade de posições onde buscar estas k-ocorrências.

A seguir trataremos do caso A. O caso B será visto mais adiante, na seção 4.2.9.

4.2.4 Caso A

Como dissemos, nosso objetivo inicial é reduzir o número de colunas candidatas a inícios de k-ocorrências. Na obtenção destas colunas ainda determinamos intervalos de linhas, dentro das colunas, onde estes inícios de k-ocorrências podem existir. Para obtermos a complexidade desejada, trataremos as escalas k uma de cada vez. Utilizaremos procedimentos diferentes para escalas menores ou iguais a m (caso 1) e para escalas maiores que m (caso 2).

Este caso envolve os seguintes passos:

1. Determinação de listas de finais de k-blocos.
2. Pré-processamento adicional.
3. Busca do padrão no texto.

Antes de descrever o conteúdo de cada passo vamos apresentar alguns conceitos utilizados nas descrições.

Inicialmente vamos dividir as linhas de cada coluna potência c em *grupos de linhas* de $km/2$ linhas sucessivas cada, o último grupo podendo conter menos linhas.

Para cada grupo $[w_1, \ldots, w_2]$, onde w_1 é a linha inicial e w_2 é a linha final, seja r_1 o mínimo $D_r[i, c]$ entre todas as linhas i em $[w_1, \ldots, w_2]$ e seja i_1 em $[w_1, \ldots, w_2]$ a linha onde este mínimo ocorre, isto é, $D_r[i_1, c] = r_1$. A posição $[i_1, c + r_1]$ é chamada de *âncora*.

Considerando uma escala k, ao dividirmos as linhas de uma coluna potência c em grupos de linhas e obtermos as respectivas âncoras, podemos determinar em quais colunas, à esquerda de c, podem ocorrer finais de k-blocos. Cada âncora determina uma seqüencia de colunas.

Conforme os valores de k e r_1 teremos alguns casos a analisar, descritos mais adiante. Para exemplificar, vamos observar o que ocorre no caso 1.1, onde $k \leq m$ e $r_1 \leq (k-1)m$. Para um grupo $[w_1, w_2]$, com âncora $[i_1, c+r_1]$, onde $c+r_1 \leq c+(k-1)m$ e i_1 pertence ao grupo $[w_1, w_2]$, observemos que, qualquer ocorrência de P_k nas colunas $c-m+1, \ldots, c$ do texto que contém a posição $[i_1, c + r_1]$, deve começar na coluna j, onde $c + r_1 - j$ é um inteiro múltiplo de k. Isto é verdade pois $c + r1$ é a posição onde ocorre uma mudança de símbolo e cada símbolo de P repete-se k vezes em cada linha de P^k.

Para cada coluna j da seqüência determinada por uma âncora, inserimos em uma lista, que denominaremos I_j^k, os índices inicial e final do grupo de linhas imediatamente anterior àquele que contém a âncora. Estes índices de linhas constituem um *intervalo de linhas* na coluna j com escala k.

Podemos ver que desta forma, reduzimos o número de colunas que podem conter k-ocorrências. Para cada coluna, temos uma indicação dos intervalos que podem conter o início de k-ocorrências. Nestes intervalos determinaremos finais de k-blocos que direcionarão a determinação das k-ocorrências, como pudemos ver no algoritmo seqüencial sublinear para busca bidimensional sem escala (seção 3.4), onde os blocos é que direcionam esta busca.

Conforme o valor da escala k, teremos dois casos a analisar.

Caso 1. $k \leq m$.

Dentro deste primeiro caso, temos 3 subcasos descritos a seguir, que são obtidos conforme os valores de r_1.

Caso 1.1. $r_1 \leq (k-1)m$.

Este caso implica que, em qualquer possível início de um k-bloco nas colunas $c-m+1, \ldots, c$ e linhas $w_1 - km/2, \ldots, w_1 - 1$, existe uma mudança de símbolo na posição $c + r_1$ e um k-bloco começando em uma das colunas $c-m+1, \ldots, c$ tem que começar em uma posição consistente com a coluna na qual o símbolo mudou. Se $r_1 = ak + b$ para $1 \leq b \leq k$ então as colunas onde P^k pode começar são $j = (c+b) - k, (c+b) - 2k, \ldots, (c+b) - \left\lfloor \frac{m+b-1}{k} \right\rfloor k$. Assim,

$$\text{para } \; j = (c+b) - k, (c+b) - 2k, \ldots, (c+b) - \left\lfloor \frac{m+b-1}{k} \right\rfloor k,$$

adicionar o intervalo $\left[w_1 - \frac{km}{2}, \ldots, w_1 - 1\right]$ a I_j^k.

Caso 1.2. $(k-1)m < r_1 \leq km$

As colunas j onde podem existir k-ocorrências estão no intervalo $c + r_1 - km < j < c$ e como $c + r_1 - km \geq c - m + 1$, para estes valores de r_1, não haverá k-ocorrências na coluna j com $c - m + 1 \leq j \leq c + r_1 - km$. Assim, o valor de r_1 não é relevante para estas colunas e iremos usar o deslocamento à esquerda descrito no próximo passo para obter as listas de intervalos. Para as colunas restantes, ou seja,

$$\text{para todo } \; j = (c+b) - k, (c+b) - 2k, \ldots, c + r_1 - km + 1,$$

adicionar o intervalo $\left[w_1 - \frac{km}{2}, \ldots, w_1 - 1\right]$ a I_j^k.

Caso 1.3. $r_1 > km$.

Seja l_2 o mínimo $D_l[i, c]$ para todo i no intervalo $[w_1, w_2]$ e seja i_1 uma linha em $[w_1, w_2]$ para a qual este mínimo é atingido. Tomemos $l_1 = l_2 - 1$.

Se $l_1 < m$ então, para todo $j = (c - l_1), c - l_1 - k, c - l_1 - 2k, \ldots, c - l_1 - \left\lfloor \frac{m - l_1 - 1}{k} \right\rfloor k$, adicionar o intervalo $\left[w_1 - \frac{km}{2}, \ldots, w_1 - 1\right]$ a I_j^k.

Se $l_1 > m$ então não há mudança de símbolo nesta área de trabalho e ela não pode contribuir com qualquer I_j^k.

A justificativa da validade deste passo para o caso 1 reside na observação de que nenhum intervalo que contenha o início de uma k-ocorrência é perdido. Dada uma k-ocorrência em alguma posição $[i, j]$, tomamos a coluna potência mais à esquerda que a intersecta. Um dos grupos de linhas ($km/2$ linhas consecutivas) da k-ocorrência deve estar contido nela. No caso A, toda seqüência de $m/2$ linhas consecutivas do padrão deve ter pelo menos uma linha de comprimento fatorado maior que 1. Tal linha fornecerá uma âncora. A âncora implicará na adição de um intervalo em I_j^k. Este intervalo deve conter i.

Para este caso, o tempo para construir as listas I_j^k é dado a seguir. Para cada coluna potência c e para cada escala k, $1 \leq k \leq m$, temos $O(N/km)$ grupos de linhas. Em cada um destes grupos, são atualizadas no máximo m/k listas. A obtenção de r_1 e l_1 é feita em tempo constante utilizando consultas de mínimo intervalar. Assim, para cada coluna potência o tempo é $O(\sum_{k=1}^{m}(N/km)(m/k)) = O(N)$. Como são $\lfloor N/m \rfloor$ colunas potência, o tempo total para construirmos estas listas para todas as escalas k, $1 \leq k \leq m$, é $O(N^2/m)$.

Caso 2. $k > m$. A principal razão para escolher as colunas potência à distância m uma da outra é que todo k-bloco intersectará uma coluna potência, mesmo para $k = 1$. Entretanto, para escalas maiores, distâncias maiores entre cada par de colunas potência são suficientes para a construção dos intervalos I_j^k. A determinação destes intervalos, para $k \geq m$ é, basicamente, a mesma para valores de k menores que m, porém utilizando-se menos colunas potência. Na tabela a seguir mostramos as faixas de intervalos de escala e as correspondentes colunas potência necessárias.

Escala	Colunas Potência
$k = 1, 2, \ldots, m-1$	$m, 2m, 3m, \ldots$
$k = m, m+1, \ldots, m^2-1$	$m^2, 2m^2, 3m^2, \ldots$
$k = m^2, m^2+1, \ldots, m^3$	$m^3, 2m^3, 3m^3, \ldots$
$\vdots$	$\vdots$

O número de faixas de escalas descritas é $\log N / \log m$. Para a primeira faixa vimos que o tempo para se obter as listas é $O(N^2/m)$. Para as escalas na segunda faixa, para cada coluna potência, temos N/km grupos de linhas em que são atualizadas no máximo m^2/k listas. Para cada coluna potência o tempo é $O(\sum_{k=m}^{m^2-1}(N/km)(m^2/k)) = O(Nm(1/m)) = O(N)$. Como existem N/m^2 colunas potência nesta faixa, o tempo total para construir as listas é $O(N^2/m^2)$. O tempo para as demais faixas é obtido de maneira análoga. Assim, o tempo para cada faixa x, $1 \leq x \leq \log N / \log m$, é $O(N/m^x)$. O tempo para todos as faixas é

$$O(\sum_{x=1}^{\log N/\log m} N^2/m^x) = O(N^2/m).$$

Ao final desta etapa, para cada coluna j e escala k, teremos uma lista de intervalos que contém os candidatos a k-ocorrência. Assim, para cada escala k são construídas no máximo $O(\lfloor \frac{N}{k} \rfloor)$ listas de intervalos.

4.2.5 Listas de Finais de k-blocos

A partir de cada lista I_j^k de intervalos, para a coluna j e escala k, construímos listas, que denominaremos Q_j^k, contendo os índices nas linhas da coluna j que são finais de k-blocos. Estas listas é que serão utilizadas para determinar as k-ocorrências na coluna j. Para construí-las, utilizaremos árvores k-altura Cartesianas construídas para cada *segmento* da coluna j. O conceito de segmento está definido a seguir.

Seja $[i_1, \ldots, i_2]$ um intervalo de linhas em uma coluna j, para $1 \leq i_1 \leq i_2 \leq N$ e $1 \leq j \leq N$. Uma *extensão* deste intervalo é o intervalo $[i_1, \ldots, \min(i_2 + km/2, N)]$, para uma escala k. Um *segmento* compreende a união de extensões adjacentes.

Em cada lista I_j^k pode haver mais de um segmento. O número de linhas entre o fim de um segmento e o início do próximo segmento é maior que $km/2$.

Para cada segmento $[v, \ldots, w]$ da coluna j e escala k, construímos as árvores k-altura Cartesianas de $B_l[v, \ldots w; c]$ e $B_r[v, \ldots w; c]$, onde c é a coluna potência relativa a j. Na construção destas árvores consideramos os valores de B_l menores que $c - j$ ou os valores de B_r menores que $km - c + j$, para caracterizar o fim de um k-bloco.

Estas árvores são obtidas em tempo $O((w - v + 1)/k)$ utilizando-se consultas do mínimo intervalar, onde cada consulta fornece a última linha de um k-bloco. Estas árvores são intercaladas em uma lista Q_j^k, e armazenam as linhas i que são finais de k-blocos na coluna j.

Para uma escala k, o número de linhas finais de k-blocos que são realmente determinados é $O(N^2/k^2)$, pois são N/k listas de intervalos e em cada lista são obtidas no máximo N/k linhas finais de bloco.

As listas Q_j^k são compostas dos k-blocos que serão usados na etapa de determinação de k-ocorrências.

Nem todos os k-blocos são obtidos. Porém, os k-blocos que estamos interessados são aqueles de altura no mínimo k. Garantimos que em uma seqüência de k-blocos de altura no mínimo k, todos eles estarão presentes na lista Q_j^k, a menos, possivelmente, do último k-bloco desta seqüência.

A justificativa para esta garantia esta descrita a seguir. Vamos considerar uma seqüência de k-blocos consecutivos $B_{j_1}, \ldots, B_{j_\alpha}$ de altura no mínimo k. Sejam $i_1, \ldots, i_\alpha$ as respectivas linhas finais dos blocos. Estas linhas estão corretamente listadas, com exceção, possivelmente, da linha final do bloco B_{j_α}.

Vamos supor que a linha final i_x de um k-bloco B_{j_x}, com $1 < x < \alpha - 1$ não seja inserida na lista, mas que as linhas finais i_{x-1} e i_{x+1} estejam corretamente inseridas. Como i_x não foi inserida, temos dois casos: $B_l[i_x, c] \geq c - j$ e $B_r[i_x, c] \geq km - c + j$, mas isto não ocorre pois esta linha é uma linha final de bloco; ou, a linha i_x não faz parte da árvore k-altura Cartesiana pois $i_{x+1} - i_{x-1} < k$, porém isto também não ocorre, uma vez que todos os blocos têm altura mínima k. Portanto, o bloco B_{j_x} estará na lista.

A linha final i_1 do primeiro k-bloco da seqüência está corretamente listado pois ele tem altura no mínimo k e os blocos anterior e posterior a ele estão a uma distância no mínimo k de sua linha final e, portanto, fará parte da árvore k-altura Cartesiana. Por motivos análogos, o k bloco de índice $j_{\alpha-1}$ também estará corretamente listado.

A linha final i_α do último k-bloco poderá não estar listada pois qualquer linha $i_\alpha + \varepsilon$, com $1 \leq \varepsilon \leq k - 1$, que seja linha final de um k-bloco e seja inserida na árvore k-altura Cartesiana evita que a linha i_α seja inserida, uma vez que a distância entre elas será menor que k.

4.2.6 Pré-Processamento Adicional

O pré-processamento adicional para este problema é o mesmo da seção 3.4.3 considerando a ordenação lexicográfica das listas Q com i primeiro, j segundo e k terceiro. O procedimento *Compara* deve ser modificado de forma que a comparação entre blocos leve em conta a escala k. A comparação de duas subcadeias é feita em tempo constante pois todas as linhas de FP^k, para toda escala k, aparecem na árvore de sufixos ST.

4.2.7 Busca do Padrão no Texto

A descrição desta etapa é basicamente a mesma da seção 3.4.3, considerando cada escala k separadamente. No lugar das listas Q_j usamos as listas Q_j^k para percorrer os blocos de texto. Durante o percurso por entre os blocos de texto precisamos fazer ainda algumas verificações:

1. Devemos verificar se o valor i do índice da linha armazenado no nó corrente e o valor da linha em seu predecessor na lista Q_j^k pertencem a um mesmo segmento. Caso isto não ocorra, existe uma lacuna entre eles e não pode haver uma ocorrência de P^k.

2. Devemos verificar se não existem blocos escondidos entre $i-k$ e i.

Se qualquer uma das condições ocorrer, avança-se para verificar se existe uma k-ocorrência iniciando-se no próximo bloco da lista Q_j^k. No item 1, basta verificar a distância entre os blocos. No item 2, utilizam-se consultas de mínimo intervalar. Em ambos os casos o tempo de verificação é constante.

O algoritmo tem funcionamento análogo ao procedimento *Busca_$\hat{F}R$* (página 84) e está descrito a seguir.

PROCEDIMENTO: Busca_$\hat{F}R$_Escala
Entrada: a cadeia C e a seqüência $\hat{F}R$.
Saída: uma lista R contendo as posições em T e as respectivas escalas k onde existe uma k-ocorrência de $\hat{F}R$.
$\{\ \hat{F}R = (FP_{l_2}, r_2)\ldots(FP_{l_{\alpha-1}}, r_{\alpha-1}).\ \}$

```
1. para k ← 1 até ⌊n/m⌋ faça
2.    para j ← 1 até n − km + 1 faça
3.       linha[0] ← 0
4.       q ← Q[j][k]
5.       i ← q.linha + 1
6.       kk ← 1
7.       cont ← 1
8.       linha[cont] ← q.linha
9.       enquanto i < n e Próximo(q)¬fim faça
10.         se Testa_itens()
11.            enquanto kk > 1 e ¬Compara(i, kk + 1, j, Próximo(q), kr_{kk+1}) faça
12.               kk ← π[kk]
13.            se Compara(i, kk + 1, j, Próximo(q), kr_{kk+1})
14.               kk ← kk + 1
15.         cont ← cont + 1
16.         q ← Próximo(q)
17.         i ← q.linha + 1
18.         linha[cont] ← q.linha
19.         se kk = α − 1
20.            Insere((linha[cont − kk + 1] − kr_1 + 1, j), R)
21.            kk ← π[kk]
22. devolva R
```

fim procedimento

As k-ocorrências encontradas são aquelas de $\hat{FR}$ em T e estas estão armazenadas em uma lista R. No procedimento *Testa_itens* são feitas as verificações de existência de lacunas e da existência de blocos escondidos, descritas anteriormente, levando tempo constante. Seja i a última linha desta ocorrência. Para cada uma destas k-ocorrências precisamos verificar se elas se estendem a k-ocorrências do primeiro e do último blocos de FR. Para isto o bloco anterior ao início da ocorrência é comparado com o primeiro bloco de FR. O bloco posterior à ocorrência, que está armazenado em Q_j^k não é necessariamente o bloco a ser comparado, pois, como vimos, o último bloco de uma seqüência de blocos de altura k pode não estar listado em Q_j^k. Se entre o fim do último bloco da ocorrência e o fim do bloco posterior à ocorrência não existir nenhum outro bloco – isto é feito em tempo constante através de consultas de mínimo intervalar em B_l e B_r – ele será o bloco a ser comparado com o último bloco de FR. Entretanto, se entre eles houver algum bloco, devemos verificar se existe o fim de um bloco entre as linhas $i+1$ e $i+kr_\alpha-1$. Se não existir, a linha $i+r_\alpha$ será considerada a linha final do bloco a ser comparado com o último bloco de FR; se existir, esta k-ocorrência deve ser descartada. Caso a k-ocorrência possa ser estendida, a posição é atualizada na lista R; caso contrário ela é removida da lista.

4.2.8 Tempo e Corretude - Caso A

A corretude do algoritmo é baseada na fato de que qualquer seqüência de k blocos de altura no mínimo k estar corretamente armazenada na lista Q_j^k, para alguma coluna j e alguma escala k, a menos do último bloco. Como a determinação das k-ocorrências é feita para cada escala k e utiliza basicamente o procedimento *Busca_*$\hat{FR}$ (página 84) com o procedimento *Compara* adaptado para lidar com escalas, as k-ocorrências, que poderão vir a ser k-ocorrências de FR em T, estarão corretamente listadas.

As k-ocorrências de FR em T estarão corretamente incluídas na lista R, pois as k-ocorrências de $\hat{FR}$ estão corretamente incluídas e somente aquelas que puderem ser estendidas a ocorrências de FR em T serão atualizadas. Estas atualizações dependem apenas de comparações entre o bloco imediatamente anterior à k-ocorrência e o primeiro bloco de FR e da obtenção do bloco posterior, que pode não estar listado, e sua comparação com o último bloco de FR.

O tempo para a etapa de pré-processamento é obtido de forma similar à análise de tempo descrita na seção 3.4.4. A seqüência C_b é composta por FT_b concatenado com $(FP_j^k)_b$, para $1 \leq j \leq m$ e $1 \leq k \leq \lfloor \frac{N}{m} \rfloor$. O comprimento desta cadeia é

$$|FT_b| + \sum_{j=1}^{m} \sum_{k=1}^{\lfloor N/m \rfloor} |(FP_j^k)_b|$$

que é no máximo $O(N^2)$, pois

$$|FT_b| \leq O(N^2)$$

e

$$\sum_{j=1}^{m}\sum_{k=1}^{\lfloor N/m\rfloor}|(FP_j^k)_b| \leq m\sum_{k=1}^{\lfloor N/m\rfloor} m(\log k+1) \leq Nm\log\lfloor\frac{N}{m}\rfloor = O(N^2).$$

A construção da árvore de sufixos é feita em tempo no máximo $O(N^2)$ e o pré-processamento para consultas LCA no mesmo tempo, pois ambos dependem do comprimento da cadeia de entrada.

A construção dos vetores B_l, B_r, D_l e D_r leva tempo $O(N^2/m)$ cada um. O pré-processamento para consultas de mínimo intervalar leva tempo $O(N)$ para um vetor de tamanho N como são $4\lfloor N/m\rfloor$ vetores de comprimento N, o tempo total para este pré-processamento é $O(N^2/m)$.

A etapa de análise do padrão leva tempo $O(|FP|)$ $(\leq O(N^2))$.

A obtenção das listas de intervalos I_j^k para cada coluna j e todas as escalas k leva tempo $O(N^2/m)$ como pode ser visto na seção 4.2.4.

Para cada escala k, temos $O(N/k)$ listas de intervalos. O comprimento total de todas os segmentos é no máximo $O(N^2/k)$. Como o tempo para construir cada árvore k-altura Cartesiana é proporcional à razão entre o comprimento do intervalo e a escala k, o tempo total para obter todas as listas Q_j^k para uma escala k é $O(N^2/k^2)$. Assim, para todas as escalas k, todas as listas Q_j^k são obtidas em tempo

$$O(\sum_{k=1}^{\lfloor N/m\rfloor} N^2/k^2) = O(N^2).$$

O pré-processamento adicional leva tempo $O(|FT| + \sum_{j,k}|Q_j^k|) = O(N^2)$.

A determinação das ocorrências depende dos tamanhos das listas e é dado por $O(\sum_{j,k}|Q_j^k|) = O(N^2)$. Portanto, todas as k-ocorrências de P em T são obtidas em tempo[1] $O(N^2)$.

4.2.9 Caso B

Vamos considerar o caso onde o padrão tem no mínimo $m/2$ linhas consecutivas de comprimento fatorado 1. Além disso, assumimos que também temos no mínimo $m/2$ colunas consecutivas do padrão de comprimento fatorado 1; caso contrário, aplicamos o algoritmo da seção 4.2.4 à rotação de 90° do padrão e do texto. Observamos que estas (no mínimo) $m/2$ linhas consecutivas devem ser todas iguais e estas (no mínimo) $m/2$ colunas consecutivas também devem ser todas iguais. Esta coleção de linhas será denominada *bloco de símbolos repetidos*. Iremos separar em dois subcasos descritos a seguir. Em ambos os subcasos, os passos a serem seguidos serão:

1. Para uma linha P_i de P convenientemente determinada, obter as k-ocorrências de P_i em todas as linhas de T. (Neste caso temos uma busca unidimensional com escala de P_i em cada linha de T.)

[1] Este tempo é mantido mesmo se a entrada não estiver na forma fatorada.

2. Algumas posições de k-ocorrência de P_i são descartadas, conforme os subcasos.

3. Para cada posição de k-ocorrência de P_i, verificamos se existe um posição de ocorrência de P em T.

Caso B.1

Existe, no mínimo, uma linha do padrão de comprimento fatorado maior que 2. Vamos mostrar como tratar deste caso.

O pré-processamento é o mesmo descrito na seção 4.2.4. Este caso especial não necessita de pré-processamento do padrão.

Seja P_i, a linha i de P de maior índice com comprimento fatorado maior que 2 que aparece acima do bloco de símbolos repetidos. O caso em que esta linha aparece abaixo é tratado simetricamente. A linha P_i tem comprimento fatorado maior que 2 e as linhas $i+d+1, \ldots, i+d+l$, $l \geq m/2$ têm comprimento fatorado 1. As linhas $i+1, \ldots, i+d$ têm comprimento fatorado 2.

O primeiro passo do algoritmo, consiste em encontrar todas as ocorrências escaladas de P_i em T. Isto pode ser feito em tempo $O(|FT|)$ pelo algoritmo para o caso unidimensional com escala da seção 2.4. Como somente um P_i^k pode aparecer em uma dada posição, existem no máximo $|FT|$ tais posições. Notemos que somente um P_i^k pode começar em um dada posição de texto. A razão é a seguinte: seja a^r a menor seqüência interna em P_i. Então a^{rk} é a menor seqüência interna em P_i^k que evita que $P_i^{k'}$, $k \neq k'$, inicie na mesma posição.

No segundo passo, descartam-se todas as posições $[l_1, l_2]$ onde as seguintes duas condições não ocorrem simultaneamente:

1. Existe um k-bloco de altura k começando em $[l_1, l_2]$ cujas linhas são todas iguais a P_i^k.

2. Existem $kl(> km/2)$ sublinhas iguais de comprimento km e comprimento fatorado 1, começando na posição $[l_1 + kd + 1, l_2]$.

A primeira condição pode ser verificada em tempo constante por consultas de mínimo intervalar em B_r e B_l, e consultas LCA com respeito a ST. A segunda condição pode ser verificada em tempo constante por consultas de mínimo intervalar em D_r, D_l, B_r e B_l.

Vamos supor que a posição $[l_1, l_2]$ passou pelas verificações anteriores. Sem perda de generalidade, supomos que a primeira mudança de símbolo em P_i ocorre na primeira metade da linha P_i. Isto significa que é impossível para qualquer uma das $k^2m^2/4$ posições $[l, l']$, com $l = l_1 + k(d+1), \ldots, l_1 + k(d+1) + km/2$ e $l' = l_2, \ldots, l_2 + km/2$, satisfazer a primeira condição. Portanto, o número total de posições restantes depois deste passo é no máximo $4N^2/k^2m^2$, para um dado k. Cada posição restante $[l_1, l_2]$ define um candidato a k-ocorrência na posição $[s_1, s_2]$, com $s_1 = l_1 - k(i-1)$ e $s_2 = l_2$.

Finalmente, no terceiro passo, obtemos as k-ocorrências. Neste passo, para cada posição candidata $[s_1, s_2]$, para cada escala k e para $0 \leq \varepsilon \leq m-1$, devemos verificar:

1. se $T[s_1 + \varepsilon k; s_2, \ldots, s_2 + km - 1] = P^k_{\varepsilon+1}$;
2. se todas as linhas $T[s_1 + \varepsilon k; s_2, \ldots, s_2 + km - 1], \ldots, T[s_1 + \varepsilon k + k - 1; s_2, \ldots, s_2 + km - 1]$ são iguais.

Se estas duas condições forem verdadeiras, a posição $[s_1, s_2]$ é o início de uma k-ocorrência. A primeira verificação pode ser feita em tempo constante de uma maneira similar ao procedimento *Compara* do caso A. A segunda verificação pode ser feita em tempo constante através de consultas de mínimo intervalar em B_l e B_r. Para cada escala k e posição candidata $[s_1, s_2]$ é necessário tempo $O(m)$ para decidir se a posição é ou não uma k-ocorrência.

O tempo total para a determinação de todas as k-ocorrências, neste caso, é dado por

$$O(|FT|) + O(\sum_{k=1}^{\lfloor \frac{N}{m} \rfloor} \frac{n^2}{k^2 m^2} m) = O(|FT|) + O(\frac{N^2}{m}) \leq O(N^2).$$

Caso B.2

Cada linha (e coluna) do padrão tem comprimento menor ou igual a 2. Existe pelo menos uma linha com comprimento igual a 2, por causa das suposições quanto ao padrão. Neste caso, as k-ocorrências são obtidas como no subcaso anterior com algumas modificações. Seja i a linha mais baixa com comprimento igual a 2 e anterior ao bloco de símbolos repetidos. Vamos, inicialmente, determinar todas as ocorrências escaladas de P_i em T.

As posições em FT onde podem haver ocorrências escaladas de P_i em T são determinadas em tempo $O(|FT|)$. Em cada posição de FT pode haver mais de uma k-ocorrência de P_i, para valores diferentes de k, no total máximo de $O(N^2)$. Seja k_{max} o maior valor de k para uma k-ocorrência de P_i em uma posição de FT. Como P_i tem comprimento escalado 2, vamos supor, sem perda de generalidade, que a mudança de símbolo ocorra na primeira metade de P_i. Assim, para cada ocorrência $[l_1, l_2]$ com escala k_{max} encontrada não poderá haver ocorrência nas posições $l_2 + k_{max}m/2, \ldots, l_2 + k_{max}m$ da linha l_1, pois nestas posições ocorre o segundo símbolo de $\bar{P}_i$. Se $k_{max} = 1$ para todas as ocorrências então o número máximo de k-ocorrências é $O(N^2/m)$. Se $k_{max} = \lfloor N/m \rfloor$ para todas as ocorrências, o número máximo de k-ocorrências é $O(N)$. Assim, no pior caso, o número máximo de k-ocorrências de P_i em T é $O(N^2/m)$. Para cada ocorrência $[l_1, l_2]$, fazemos o descarte de algumas delas utilizando as condições do subcaso anterior.

Mantendo a suposição de mudança de símbolo de P_i na sua primeira metade, o número máximo de posições l_1, l_2 onde pode haver k-ocorrências é $4N^2/k^2m^2$.

As k-ocorrências são então obtidas como no subcaso anterior em tempo

$$O(|FT|) + O(N^2/m) + O(\sum_{k=1}^{\lfloor \frac{N}{m} \rfloor} \frac{N^2}{k^2 m^2} m) = O(\frac{N^2}{m}) \leq O(N^2).$$

4.2.10 Tempo e Corretude - Caso B

O tempo para este caso foi mostrado durante a descrição de como funciona o algoritmo. A obtenção das k-ocorrências unidimensionais de P_i está correta pois utilizamos o algoritmo da seção 2.4. Ao se descartar algumas destas ocorrências, estamos descartando apenas aquelas em que não pode ocorrer o padrão P com escala k. Com isto, seguramente, estaremos mantendo posições corretas e nenhuma posição que seria de ocorrência foi erroneamente descartada. Finalmente, para cada posição candidata $[s_1, s_2]$ comparamos cada linha de texto escalado a k com a correspondente linha escalada do padrão.

4.3 O Algoritmo em CGM

O algoritmo CGM, neste caso, segue o que foi feito no caso seqüencial. As partes de pré-processamento e análise do padrão são feitas localmente em cada processador. Na etapa de análise do texto teremos que dividir em casos dependendo dos dados do padrão. A descrição desta etapa para o modelo CGM segue algumas idéias do caso sem escala da seção 3.5. Nas descrições, a ordem em que são feitas as computações dos dados das regiões de fronteira e o envio e recebimento de dados é alterado, pois precisaremos de informações extras para que a linearidade das computações locais seja mantida.

A distribuição dos dados nos processadores é a mesma do caso bidimensional descrito na seção 3.3.1. Cada processador recebe uma submatriz de ordem $n = N/\sqrt{p}$ e a matriz padrão P de ordem m. Um cuidado a ser tomado é com a faixa de valores que a escala k pode assumir. No algoritmo seqüencial, o maior valor possível para a escala k é $\lfloor N/m \rfloor$. No caso CGM, este valor é $\lfloor n/m \rfloor = \lfloor N/(m\sqrt{p}) \rfloor$.

4.3.1 Caso A em CGM

A seguir temos a descrição da etapa de análise do texto para o caso A em que qualquer seqüência de $km/2$ linhas consecutivas do padrão tem pelo menos uma linha de comprimento fatorado maior que 2.

Em cada processador, o algoritmo, primeiramente, obtém as k-ocorrências do padrão no texto localmente armazenado. Em seguida, são determinadas as posições de confirmação nas fronteiras Norte, Nordeste, Noroeste, Oeste e Sudoeste, conforme o índice do processador. Estas posições de confirmação são enviadas para o respectivo vizinho. Além destas, são enviadas para o vizinho Norte, quando este existir, informações adicionais que servirão para obter as posições candidatas. As posições candidatas são deteminadas no passo seguinte. De posse das posições candidatas e de confirmação, estas são combinadas para determinar as k-ocorrências. Podemos perceber que obtivemos, primeiro, as posições de confirmação e as enviamos para os processadores vizinhos, diferentemente dos algoritmos CGM dos capítulos anteriores onde as posições candidatas eram obtidas primeiro. A razão para esta alteração é a utilização de apenas uma rodade de comunicação para a troca de dados. Se continuássemos considerando a

mesma seqüência dos outros algoritmos, seriam necessárias duas rodadas de comunicação, como podemos acompanhar na discussão mais adiante, na página 117.

ALGORITMO: Busca_com_Escala_Bi_CGM
Entrada: uma matriz texto T, uma matriz padrão P.
Saída: listas R nos processadores com as posições da submatriz local S onde existem k-ocorrências de P.
{ Cada processador de índice $(i-1)\sqrt{p}+j-1$ recebe uma submatriz $S_{i,j}$ de ordem $n=\frac{N}{\sqrt{p}}$ da matriz T, com $1 \leq i,j \leq \sqrt{p}$ e a matriz padrão P. }

1. Cada processador h executa o caso A do algoritmo de busca bidimensional de padrões com escala para os dados locais, obtendo uma lista R.
2. Cada processador h, $0 < h \leq p-1$, executa o procedimento *Constrói_σ*.
3. Cada processador h, $0 < h \leq p-1$, executa:
   3.1 o procedimento *Fronteira_ONON*, se h mod $\sqrt{p} > 0$ e $h > \sqrt{p}$, obtendo as listas C_O. C_{SO}, C_N, C_{NE} e C_{NO}.
   3.2 o procedimento *Fronteira_N*, se h mod $\sqrt{p} = 0$, obtendo as lista C_N e C_{NE}.
   3.3 o procedimento *Fronteira_O*, se $h < \sqrt{p}$, obtendo as listas C_O e C_{SO}
4. Cada processador h, $0 < h \leq p-1$, envia:
   4.1 as listas: C_O e C_{SO} para o processador $h-1$, que recebe em C_O e C_{SO}, respectivamente; C_N e C_{NE} para o processador $h-\sqrt{p}$, que recebe em C_N e C_{NE}, respectivamente; e C_{NO} para o processador $h-\sqrt{p}-1$, que recebe em C_{NO}. Onde h mod $\sqrt{p} > 0$ e $h > \sqrt{p}$. Também são enviados os vetores $\bar{l}$, $\bar{r}$, l_1 e r_1. (Ver descrição da obtenção da posições de confirmação da fronteira Norte a seguir.)
   4.2 as listas C_N e C_{NE} para o processador $h-\sqrt{p}$, que recebe em C_N e C_{NE}, respectivamente, onde h mod $\sqrt{p} = 0$. Também são enviados os vetores $\bar{l}$, $\bar{r}$, l_1 e r_1. (Ver descrição da obtenção da posições de confirmação da fronteira Norte a seguir.)
   4.3 as listas C_O e C_{SO} para o processador $h-1$, que recebe em C_O e C_{SO}, respectivamente, onde $h < \sqrt{p}$.
5. Cada processador h, $0 \leq h < p-1$,executa:
   5.1 o procedimento *Fronteira_SSEE*, se $(h+1)$ mod $\sqrt{p} > 0$ e $h < p-\sqrt{p}$, obtendo a lista C_{SSEE}.
   5.2 o procedimento *Fronteira_S*, se $(h+1)$ mod $\sqrt{p} = 0$, obtendo a lista C_S.
   5.3 o procedimento *Fronteira_E*, se $h \geq p-\sqrt{p}$, obtendo a lista C_E.
6. Cada processador h, $0 \leq h < p-1$, executa o procedimento *Combina_Bi_Escala*.

fim algoritmo

Como consideramos cada escala k separadamente, as delimitações das fronteiras depende do valor de k. Assim, as fronteiras norte e sul são faixas horizontais $[1,\ldots,km-1;1,\ldots,n]$ e $[n-km+2,\ldots,n;1,\ldots,n]$, respectivamente; as fronteiras leste e oeste são faixas verticais $[1,\ldots,n;n-km+2,\ldots,n]$ e $[1,\ldots,n;1,\ldots,km-1]$, respectivamente; as fronteiras nordeste, sudeste, sudoeste e noroeste são matrizes quadradas $[n-km+2,\ldots,n;1,\ldots km-1]$, $[n-km+2,\ldots n;n-km+2,\ldots,n]$, $[1,\ldots,km-1;n-km+2,\ldots,n]$ e $[1,\ldots,km-1;1,\ldots km-1]$, respectivamente.

As listas Q_j^k são obtidas normalmente para os dados locais não pertencentes às regiões fronteiriças. Para estas regiões, a obtenção destas listas apresenta algumas particularidades.

Vamos lembrar que na construção das listas I_j^k são utilizados os vetores D_l e D_r. As listas Q_j^k são obtidas a partir destas listas de intervalos e dos vetores B_l e B_r. Estas listas são utilizadas para as buscas nas posições não pertencentes às regiões de fronteira. Os vetores D_l, D_r, B_l e B_r são construídos para as colunas potência que são usadas como referência.

Para as fronteiras Leste e Oeste, as colunas a serem usadas como referências serão a última e a primeira, respectivamente.

No caso da fronteira Leste, as listas I_j^k e Q_j^k, para $n-km+2 \leq j \leq n$ e $1 \leq k \leq \lfloor n/m \rfloor$, são obtidas como descrito a seguir. Como desconhecemos os dados do vizinho Leste, consideramos como se ocorresse o caso 1.3 da seção 4.2.4, e as listas de intervalos são obtidas como naquele caso. As listas Q_j^k são obtidas usando estas listas de intervalos e o vetor B_l na última coluna. Este vetor B_l é obtido como na versão CGM do algoritmo sublinear na seção 3.5. A árvore k-altura Cartesiana é construída para estes vetores armazenando as linhas cujos valores em B_l sejam menores que $n-j+1$.

Para a fronteira Oeste, onde $1 \leq j \leq km-1$, a construção é similar. As listas I_j^k são obtidas considerando o simétrico do caso 1.3 da seção 4.2.4, isto é, supomos que $l_1 > km$ ($r_1 > km$ no caso original) e as listas são obtidas de acordo com os valores de r_1 (l_1 no caso original). As listas Q_j^k são então obtidas utilizando-se as listas I_j^k e o vetor B_r na primeira coluna. O vetor B_r é obtido como na seção 3.5 e a árvore k-altura Cartesiana para este vetor armazena as linhas cujos valores em B_r sejam menores que j. As listas armazenam as linhas de início dos k-blocos e seus valores estão em ordem decrescente.

As listas Q_j^k para os dados na fronteira Norte dependem da forma como as colunas potência são divididas em grupos de linhas. Como cada grupo de linhas tem $km/2$ linhas, o número de grupos é $2n/km$. Vamos definir ϱ o resto da divisão $2n/km$. Temos então dois casos, que dependem do valor de ϱ, descritos a seguir.

Caso 1. $\varrho = 0$, isto é, todos os grupos de linhas têm a mesma quantidade de linhas. Neste caso, as listas Q_j^k construídas para os dados não pertencentes às regiões de fronteira são utilizados a partir do elemento que aponta para o último elemento cujo valor da linha armazenado esteja dentro da fronteira Norte. Esta lista é percorrida a partir deste nó em direção ao início da lista.

Caso 2. $\varrho > 0$, isto é, o último grupo de linhas é formado por menos de km/2 linhas, mais exatamente, ϱ linhas. Para cada coluna potência c, a restrição desta coluna à fronteira Norte

apresenta 3 intervalos: $[1,\ldots,km/2-\varrho]$, $[km/2-\varrho+1,\ldots,km-\varrho]$ e $[km-\varrho+1,\ldots,km]$. As listas de intervalos são construídas baseadas no segundo e no terceiro intervalos e conterão o primeiro e/ou o segundo intervalos. As listas Q_j^k são então construídas a partir destas listas de intervalos e dos vetores B_r e B_l nas colunas potência.

Para os dados na fronteira Nordeste, as listas Q_j^k são obtidas combinando-se as idéias descritas para as fronteiras Norte e Leste. Para os dados na fronteira Noroeste são combinadas as idéias utilizadas nas fronteiras Norte e Oeste.

Para a determinação das posições candidatas na fronteira Sul são necessárias algumas informações que vêm do vizinho Sul. Lembremos que para construir as listas Q_j^k precisamos obter os intervalos que possivelmente conterão os inícios das k-ocorrências. Cada intervalo é inserido com base na análise do intervalo imediatamente abaixo dele.

Para cada coluna potência c suas linhas são divididas em grupos de linhas com $km/2$ linhas cada. Na fronteira Sul, analisamos as últimas km linhas para determinar as posições candidatas. Lembrando que o número de grupos é $\lceil 2n/km \rceil$ e, considerando ϱ o resto da divisão $2n/km$, temos dois casos a analisar:

Caso 1. $\varrho = 0$, isto é, o número de linhas de todos os grupos é igual a $km/2$. Neste caso, as últimas $km/2$ linhas eventualmente incluirão o intervalo $[n-km+1,\ldots,n-km/2]$ em algumas listas I_j^k. Para analisar as últimas $km/2$ linhas como possíveis inícios de k-ocorrências é necessário verificar o que ocorre no intervalo seguinte que compreende as primeiras $km/2$ linhas do vizinho Sul. Para isto, considerando cada coluna potência c, os vetores de mínimos intervalares $\bar{l}$ e $\bar{r}$ entre as posições 1 e $km/2$ dos vetores $D_l[c]$ e $D_r[c]$, respectivamente, são calculados e enviados juntamente com as posições de confirmação obtidas na fronteira Norte do vizinho Sul. Se os valores recebidos forem iguais a l_1 e r_1 do último grupo em cada coluna potência, a árvore k-altura Cartesiana construída para o penúltimo grupo já contempla estas posições. Caso contrário, será necessário a construção da árvore k-altura Cartesiana para o último grupo, obtendo, assim, as listas Q_j^k.

Caso 2. $\varrho > 0$. As linhas entre $n-km+1$ e $n-\varrho-km/2$ pertencem a um intervalo cuja inclusão ou não depende do intervalo imediatamente seguinte que está inteiramente contido no processador. A inclusão do intervalo $[n-\rho-km/2+1,\ldots,n-\rho]$ depende dos valores l_1 e r_1 obtidos através do mínimo intervalar, para cada coluna potência c, dos vetores $D_l[c]$ e $D_r[c]$, respectivamente, no intevalo seguinte. Este intervalo é formado pelos intervalos $[n-\varrho+1,\ldots,n]$ no processador local e $[1,\ldots,km/2-\varrho]$ no vizinho Sul. Assim, é necessário uma consulta de mínimo intervalar para intervalos em processadores vizinhos, que é feita utilizando uma comunicação, como descrito na seção 1.8.3, onde são enviados, do vizinho Sul, os valores de $\bar{l}$ e $\bar{r}$, para cada coluna potência, dos mínimos intervalares de D_l e D_r no intervalo $[1,\ldots,km/2-\varrho]$. Para a inclusão do intervalo $[n-\varrho+1,\ldots,n]$ serão necessários, para cada coluna potência, os valores de l_1 e r_1 obtidos dos vetores D_l e D_r no intervalo $[km/2-\varrho+1,\ldots,km-\varrho]$ no vizinho Sul. Todos estes valores são obtidos durante a determinação das posições de confirmação na fronteira Norte do vizinho Sul. Na comunicação são enviados, do vizinho Sul, as posições de confirmação da fronteira Norte, e

para cada coluna potência, os valores de $\bar{l}$ e $\bar{r}$, relativas ao intervalo $[1, \ldots, km/2 - \varrho]$, e l_1 e r_1 relativas ao intervalo $[km/2 - \varrho + 1, \ldots, km - \varrho]$. De posse destes valores, os intervalos I_j^k para a fronteira Sul podem ser construídos e, a partir deles juntamente com os vetores B_l e B_r, é possível obter as listas Q_j^k.

As listas Q_j^k para os dados nas fronteiras Sudeste são obtidos a partir das idéias contidas na obtenção das listas nas fronteiras Sul e Leste, e Sul e Oeste, respectivamente. Para a fronteira Sudeste são enviados os valores de $\bar{l}$, $\bar{r}$, l_1 e r_1 do vetor $D_l[n]$.

Para a construção das listas Q_j^k para os dados da fronteira Sudoeste são necessárias informações do vizinho Sul e após a obtenção das posições de confirmação devemos enviá-las para o vizinho Oeste. Se utilizássemos esta seqüência de passos seriam necessárias duas rodadas de comunicação. Para evitar uma rodada de comunicação a mais, as posições de confirmação da fronteira Sudoeste são obtidas de maneira um pouco diferente. Primeiramente, eliminamos a construção das listas de intervalos, com isto não precisamos mais das informações do vizinho Sul. Para cada escala k, construímos as listas Q_j^k para todas as colunas j, $1 \leq j \leq km - 1$, considerando as listas de intervalos formadas apenas pelo intervalo $n - km + 2, \ldots, n$.

A partir da obtenção das listas Q_j^k, as posições candidatas e as posições de confirmação são obtidas como na seção 3.5.2 com as adaptações necessárias à utilização de escalas. Nas listas de posições candidatas e de confirmação são também armazenadas os valores das escalas. Para as posições de confirmação nas fronteiras Nordeste e Noroeste aos valores da escala armazenados é adicionado $\lfloor n/m \rfloor$ para diferenciar das posições de confirmação nas fronteiras Norte e Oeste, como na seção 2.5.

Os valores recebidos e localmente calculados são combinados como no caso bidimensional considerando-se também as escalas.

4.3.2 Caso B em CGM

A descrição do algoritmo para este caso é similar à do algoritmo *Busca_com_Escala_Bi_CGM* (página 115). Localmente, iremos utilizar o algoritmo para o caso B. Novamente, as principais diferenças estão na obtenção dos dados nas regiões de fronteira.

Nas fronteiras Leste e Oeste, seja a linha i como no caso B.1 ou B.2, conforme o padrão. Buscamos todas as ocorrências escaladas de P_i, como na seção 2.5. A partir destas posições, obtemos as posições das k-ocorrências, para cada escala k, considerando os sub-blocos de largura k, de acordo com a fronteira.

Na fronteira Oeste, as duas primeiras posições de cada linha de FS não são testadas como fim de k-ocorrências de P_i. Por isso, é necessário o envio, para o vizinho Oeste, das informações dos símbolos e expoentes nestas duas primeiras posições em cada linha de FS, num total de $O(n)$ dados. Estes dados são enviados juntamente com as posições de confirmação. Tendo recebido os dados do vizinho Leste, eles são combinados de forma a confirmar as k-ocorrências de P_i e as k-ocorrências de P. Esta quantidade de dados pode ser diminuída se enviarmos as informações de símbolos e expoentes apenas para as linhas que contenham ocorrências de P_i e

para as linhas onde os dois primeiros símbolos da representação fatorada sejam iguais aos dois primeiros símbolos de $\bar{P}_i$ (representação fatorada da linha i de P) ou para as linhas em que o primeiro símbolo seja igual ao primeiro símbolo de $\bar{P}_i$.

Para os dados das fronteiras Norte e Sul, a descrição é mais delicada. Vamos descrever como proceder para a obtenção das posições de confirmação na fronteira Norte. As posições candidatas na fronteira Sul são obtidas de maneira simétrica. Conforme os dados do padrão vamos considerar dois casos:

Caso 1. Vamos supor que exista uma linha com comprimento fatorado maior que 1 depois do bloco de símbolos repetidos. Seja i a linha de maior índice com esta propriedade. Vamos supor que o bloco de símbolos repetidos seja formado pelas linhas $i-d-l+1, \ldots, i-d$, com $l \geq m/2$, as d linhas $i-d+1, \ldots, i$ têm comprimento fatorado maior que 1. Determinamos todas as ocorrências escaladas de P_i na submatriz $S[1, \ldots, \lfloor n/m \rfloor m; 1, \ldots, n]$. O número de possíveis ocorrências é menor que $O(n^2/m)$, como na seção 4.2.9.

Para cada posição $[l_1, l_2]$ de ocorrência, temos:

1. Se $l_1 \geq k$, verificamos:
   (a) Se existe um k-bloco de altura k terminando em $[l_1, l_2]$ cujas linhas são todas iguais a P_i^k.
   (b) Se $l_1 - kd + 1 > 1$ e existem $\min(kl, l_1 - kd)$ sublinhas iguais de comprimento km e comprimento fatorado 1, terminando na posição $[l_1 - kd, l_2]$.
2. Se $l_1 < k$, verificamos se existe um k-bloco de altura l_1 terminando em $[l_1, l_2]$ cujas linhas são todas iguais a P_i^k.

Se as condições 1.(a) e 1.(b) ambas falharem para o caso $l_1 \geq k$, a posição $[l_1, l_2]$ não é considerada. A posição $[l_1, l_2]$, para $l_1 < k$ só permanece como possível fim de ocorrência se a condição 2 for verdadeira. As posições $[l_1, l_2]$ correspondentes ao fim de uma k-ocorrência com $l_1 > km$ também são eliminadas.

Vamos supor que a primeira mudança de símbolo de P_i ocorra na primeira metade desta linha. Então, para cada ocorrência $[l_1, l_2]$ não pode haver uma ocorrência nas colunas $l_2 + km/2 + 1, \ldots, l_2 + km$. Assim, o número de possíveis ocorrências para uma escala k é $2n$, pois o número de posições na fronteira Norte é kmn e para cada posição são descartadas $km/2$ posições. Cada posição restante $[l_1, l_2]$ define uma posição $[s_1, s_2]$ de final de ocorrência, onde $s_1 = l_1 + k(m - i)$ e $s_2 = l_2$.

Para cada posição candidata $[s_1, s_2]$, para cada escala k e para $0 \leq \varepsilon \leq \min(\lfloor s_1/k \rfloor, m-1)$, devemos verificar:

1. se $T[s_1 - \varepsilon k; s_2, \ldots, s_2 + km - 1] = P^k_{m-\varepsilon}$;
2. se todas as linhas $T[s_1 - \varepsilon k - k + 1; s_2, \ldots, s_2 + km - 1], \ldots, T[\min(1, s_1 - \varepsilon k); s_2, \ldots, s_2 + km - 1]$ são iguais.

Se estas as condições dos itens forem verdadeiras a posição $[s_1, s_2]$ é o fim de uma k-ocorrência e a posição $[s1, s2 + km - 1]$ é uma posição de confirmação na fronteira Norte. A primeira verificação pode ser feita em tempo constante de uma maneira similar ao procedimento *Compara* do caso A. A segunda verificação pode ser feita em tempo constante através de consultas de mínimo intervalar em B_l e B_r. Para cada escala k e posição $[s_1, s_2]$ é necessário tempo máximo $O(m)$ para decidir se a posição é ou não uma posição de confirmação.

O tempo total para a obtenção das posições de confirmação na fronteira Norte é

$$O(n^2/m) + O(\sum_{k=1}^{\lfloor \frac{n}{m} \rfloor} 2nm) = O(n^2) = O(N^2/p).$$

Se $i = m$, não há mais nada a fazer. Caso i seja menor que m, existe um outro bloco de símbolos repetidos de altura menor que $m/2$. Devemos verificar a possibilidade deste bloco ser parte do fim de uma k-ocorrência do vizinho Norte. A determinação das posições de confirmação para este bloco são determinadas como no final do caso a seguir e tem complexidade de tempo $O(n^2) = O(N^2/p)$. Portanto, para determinar todas as posições de confirmação para este caso é necessário tempo $O(n^2) = O(N^2/p)$.

Caso 2. Se não houver uma linha de comprimento fatorado maior que 1 após o bloco de símbolos repetidos, vamos considerar a linha i de maior índice com comprimento fatorado no mínimo 2. O bloco de símbolos repetidos é formado pelas linhas $i + 1, \ldots, m$, com $l = m - i \geq m/2$. Vamos determinar todas as ocorrências escaladas de P_i na submatriz $S[1, \ldots, \lfloor n/m \rfloor m; 1, \ldots, n]$. O número de possíveis ocorrências é também, no máximo $O(n^2/m)$.

Para cada posição $[l_1, l_2]$ de ocorrência, temos:

1. Se $l_1 \geq k$, verificamos se existe um k-bloco de altura k terminando em $[l_1, l_2]$ cujas linhas são todas iguais a P_i^k. Caso contrário, se $l_1 < k$, verificamos se existe um k-bloco de altura l_1 terminando em $[l_1, l_2]$ cujas linhas são todas iguais a P_i^k.
2. Se existem kl sublinhas iguais de comprimento km e comprimento fatorado 1, terminando na posição $[l_1 + kl, l_2]$.

Se pelo menos uma das duas condições não for verdadeira, a posição $[l_1, l_2]$ é descartada. As posições $[l_1, l_2]$ correspondentes ao fim de uma k-ocorrência com $l_1 > km$ também são eliminadas.

Suporemos que a primeira mudança de símbolos em P_i ocorra na primeira metade desta linha. Então, para cada ocorrência $[l_1, l_2]$, não pode haver ocorrências nas colunas $l_2 + km/2 + 1, \ldots, l_2 + km$. O número máximo de ocorrências para uma escala k é n, pois o número de posições onde pode haver ocorrências é $\frac{km}{2}n$ e para cada posição não existem ocorrências em $km/2$ posições. Os finais de possíveis ocorrências são as posições $[s_1, s_2]$, com $s_1 = l_1 + kl$ e $s_2 = l_2$.

Para cada posição $[s_1, s_2]$, para cada escala k e para $0 \leq \varepsilon \leq \min(\lfloor s_1/k \rfloor, m - 1)$, devemos verificar:

1. se $T[s_1 - \varepsilon k; s_2, \ldots, s_2 + km - 1] = P^k_{m-\varepsilon}$;
2. se todas as linhas $T[\min(1, s_1 - \varepsilon k - k + 1); s_2, \ldots, s_2 + km - 1], \ldots, T[s_1 - \varepsilon k; s_2, \ldots, s_2 + km - 1]$ são iguais.

Se estas as condições dos itens forem verdadeiras a posição $[s_1, s_2]$ é o fim de uma k-ocorrência e a posição $[s1, s2 + km - 1]$ é uma posição de confirmação na fronteira Norte. A primeira verificação pode ser feita em tempo constante de uma maneira similar ao procedimento *Compara* do caso A. A segunda verificação pode ser feita em tempo constante através de consultas de mínimo intervalar em B_l e B_r. Para cada escala k e posição candidata $[s_1, s_2]$ é necessário tempo $O(m)$ para decidir se a posição é ou não uma k-ocorrência.

O tempo total para a obtenção destas posições de confirmação é

$$O(\lfloor n/m \rfloor |FT|) + O(\sum_{k=1}^{\lfloor \frac{n}{m} \rfloor} nm) = O(n^2) = O(N^2/p).$$

Resta ainda verificar as ocorrências do bloco de símbolos repetidos. Para isto, determinamos as ocorrências escaladas $[l_1, l_2]$ de P_m. Como pode ocorrer mais de uma escala em cada posição, armazenamos apenas o maior valor possível k' da escala para esta posição. Temos então no máximo $O(n^2)$ posições. As posições nas quais $k'l < l_1$ são eliminadas. Para cada posição restante obtemos os valores dos mínimos intervalares nos vetores B_l e B_r entre as linhas 1 e $l_1 - 1$. Com isto sabemos se existe o fim de um bloco neste intervalo de linhas, e se existir podemos determinar o valor máximo k'' para o qual as linhas $1, \ldots, l_1$ formam um k''-bloco. Se $k''l < l_1$, a posição é eliminada. Caso contrário, a posição $[l_1, l_2]$ indica um final de um k-bloco com escala $k = \min(k', k'')$. A posição será uma posição de confirmação com o valor k armazenado e com uma indicação de que este é o valor máximo da escala para uma k-ocorrência terminando nesta posição.

Como as consultas de mínimo intervalar são feitas em tempo constante, estas posições de confirmação são obtidas em tempo $O(n^2) = O(N^2/p)$.

Portanto, todas as posições candidatas na fronteira Norte, para este caso, são obtidas em tempo máximo $O(n^2) = O(N^2/p)$.

Em ambos os casos, todas as posições de ocorrência com as respectivos valores da escala (ou escala máxima e indicador) são obtidos em tempo $O(n^2) = O(N^2/p)$.

As posições de confirmação são enviadas para o vizinho Norte. No vizinho Norte os dados recebidos incluem as posições de confirmação como nos casos sem escala do capítulo anterior e também posições de confirmação que contém o valor máximo da escala e que se encontra na mesma coluna de uma possível candidata. A distinção na manipulação destes dois casos é feita no procedimento *Combina_Bi_Escala* conforme a presença do indicador.

Como já dissemos, as posições candidatas da fronteira Sul são obtidas de maneira simétrica. As posições candidatas na fronteira Sudeste e as posições de confirmação nas fronteiras Sudeste, Nordeste e Noroeste são obtidas combinando-se as descrições vistas.

De posse dos dados recebidos e dos calculados localmente, as k-ocorrências são obtidas combinando-se estas informações como na seção 3.5.2 considerando-se também as escalas.

4.3.3 Tempo e Corretude

A corretude do algoritmo é decorrente da corretude do algoritmo seqüencial e da corretude do algoritmo CGM para o caso sublinear da seção 3.5.2. Os tempos para os casos A e B são analisados a seguir.

Caso A

No caso A, as k-ocorrências para os dados locais são obtidas em tempo $O(n^2) = O(N^2/p)$ conforme vimos na seção 4.2.8. Vamos analisar o tempo na obtenção das posições candidatas e de confirmação nas posições de fronteira.

Na obtenção das listas de intervalos da fronteira Leste, para cada escala k, são $O(n/km)$ grupos de linhas para a última coluna. Em cada grupo, são atualizadas no máximo $km/k = m$ listas. Assim é necessário tempo $O(\sum_{k=1}^{\lfloor n/m \rfloor} n/k) = O(n(\log n/\log m))$ para obter todas as listas de intervalos.

Para cada escala k, temos $O(m)$ listas de intervalos. O comprimento total de todos os segmentos é no máximo $O(nm)$. Como o tempo para construir a árvore k-altura Cartesiana é proporcional à razão entre o comprimento do intervalo e a escala k, o tempo total para construir todas as listas Q_j^k é dado por

$$O(\sum_{k=1}^{\lfloor n/m \rfloor} \frac{nm}{k}) = O(\frac{\log n}{\log m} nm) \leq O(n^2),$$

pois $m/\log m$ é uma função crescente e $m \leq n$.

O pré-processamento adicional e a determinação das ocorrências depende dos tamanhos das listas e leva no máximo tempo $O(n^2)$.

Para a fronteira Oeste, o raciocínio é análogo e portanto, a obtenção das posições de confirmação, também consome tempo máximo $O(n^2)$.

Para a fronteira Norte serão incluídos no máximo 2 intervalos, para cada escala k, em no máximo n/k listas. Assim, o tempo para construir estas listas de intervalos é $O(n/k)$.

O comprimento total dos segmentos para cada escala k é no máximo $O((n/k)(km)) = O(nm)$ pois são no máximo n/k listas e cada lista com segmentos de tamanho máximo km. Analogamente ao caso da fronteira Leste, o tempo total para obter as listas Q_j^k é no pior caso $O(n^2)$. (Lembremos que se $2n/km$ é um inteiro as listas já estão calculadas.) As ocorrências são obtidas em tempo máximo $O(n^2)$.

Para a fronteira Sul, as posições candidatas são também obtidas em tempo máximo $O(n^2)$ e a justificativa é similar ao caso da fronteira Norte.

No caso da fronteira Sudeste, o tempo para construir as listas de intervalos é $O(m)$, pois são incluídos no máximo 2 intervalos em $km/k = m$ listas. Assim, é necessário tempo $O(\sum_{k=1}^{\lfloor n/m \rfloor} m) = O(n)$ para se obter todas as listas.

As listas Q_j^k para esta fronteira são obtidas em tempo máximo $O(\sum_{k=1}^{\lfloor n/m \rfloor} m^2) = O(nm) \leq O(n^2)$, pois o comprimento total máximo dos segmentos é $O(kmm)$ e, conseqüentemente, o tempo para construir as árvores k-altura Cartesianas é $O(m^2)$. A determinação das posições candidatas para esta fronteira é também no máximo $O(n^2)$.

As posições de confirmação das fronteiras Sudeste, Nordeste e Noroeste são obtidas também em tempo máximo $O(n^2)$.

Para os dados da fronteira Sudoeste a determinação do tempo é dada a seguir. Como são km colunas com km posições, o comprimento total máximo dos segmentos é k^2m^2. As árvores k-altura Cartesianas levam tempo proporcional à razão entre o comprimento total e o valor da escala. Assim, as listas, para uma escala k são construídas em tempo $O(\sum_{k=1}^{\lfloor n/m \rfloor} km^2) = O(n^2)$. O comprimento total das listas é também $O(n^2)$, pois, para cada escala k, são km listas com km/k nós. Logo, a obtenção das posições de confirmação, que depende do comprimento total das listas, leva tempo $O(n^2)$.

O procedimento *Combina_Bi_Escala* tem comportamento análogo ao procedimento *Combina_Bi* (página 71) mas considerando também as escalas para a determinação das k-ocorrências. Este procedimento é executado em tempo $O(n^2)$ uma vez que depende do tamanho das listas de posições candidatas e de confirmação.

Portanto, o algoritmo para o caso A roda em tempo linear no tamanho da entrada, isto é, $O(N^2/p)$.

No algoritmo é utilizada somente uma rodada de comunicação entre os processadores. No caso da comunicação Leste-Oeste são enviadas apenas as posições de confirmação num total de no máximo $O(n^2)$ dados. Quando a comunicação é no sentido Norte-Sul são enviados $O(n^2)$ dados relativos às posições de confirmação e também $O(n/m)$ dados relativos aos mínimos intervalares descritos nos casos 1 e 2 da página 117. No caso da comunicação Sudeste-Noroeste são enviados no máximo $O(n^2)$ dados. Como um processador pode enviar para os vizinhos Norte, Leste e Noroeste, ele estará enviando no máximo $O(n^2)$ dados. Por outro lado, um processador pode receber informações dos vizinhos Sul, Oeste e Sudeste num total de $O(n^2)$ dados. Assim, a quantidade de informações trocadas é no máximo $O(n^2) = O(N^2/p)$, que está dentro do limite imposto pelo modelo CGM.

A árvore de sufixos com o pré-processamento para consultas LCA ocupa $O(n^2)$ de memória. Os vetores B_l, B_r, D_l e D_r

O algoritmo CGM para este caso utiliza memória $O(n^2)$ no armazenamento dos dados. A etapa de construção das estrutura de dados, como no algoritmo da seção 3.5, consome este espaço. O pré-processamento dos vetores B_l, B_r, D_l e D_r para consultas de mínimo intervalar

utiliza memória $O(n)$. A análise do padrão é a mesma da seção 3.4.2.

Na obtenção das posições candidatas e de confirmação nas regiões de fronteira ocupa-se, no máximo, a mesma quantidade de memória que na obtenção das k-ocorrências nas demais posições, uma vez que os procedimentos que são utilizados nas fronteiras obedecem basicamente o mesmo algoritmo seqüencial restrito a estas regiões.

Para cada escala k, obtemos listas de intervalos ocupando espaço $O(n/k)$. O espaço para estas listas pode ser reaproveitado para valores de escala diferentes. Assim, no pior caso, quando $k = 1$, o espaço ocupado por esta lista é $O(n)$. Se este reaproveitamento não é feito ainda ocupamos espaço dentro dos limites do modelo dado por $O(\sum_{k=1}^{\lfloor n/m \rfloor} n/k) = O(n \log n / \log m)$.

As listas de finais de k-blocos devem ser mantidas durante toda a execução. Como o tamanho total das listas é $O(n^2)$, o espaço ocupado por elas é $O(n^2)$.

Em cada processador, as k-ocorrências, as posições candidatas e as posições de confirmação recebidas são armazenadas em listas cujo espaço ocupado não excede $O(n^2)$. Após, o procedimento *Combina_Bi_Escala* que utiliza estas listas. As informações estarão na lista R que ocupa espaço no máximo $O(n^2)$.

Portanto, na execução do algoritmo CGM para este caso, a memória total ocupada é $O(n^2) = O(N^2/p)$.

Caso B

Para o caso B, o tempo de processamento foi analisado durante a descrição dos passos para este caso e é no máximo $O(n^2)$. O procedimento *Combina_Bi_Escala* é o mesmo do caso A e roda em tempo $O(n^2)$.

O pré-processamento da entrada é o mesmo do caso A, ocupando memória $O(n^2)$, bem como a determinação dos vetores B_l e B_r e da estrutura de dados para consultas de mínimo intervalar . Na obtenção das posições de k-ocorrências ou na obtenção das posições candidatas e de confirmação utilizamos no máximo $O(n^2)$ que é o máximo de memória necessária para a busca unidimensional com escala e para armazenar as posições que serão analisadas. As posições que não sejam k-ocorrências são descartadas.

É necessário apenas uma rodada de comunicação onde cada processador envia as posições de confirmação para os vizinhos Norte, Leste e Noroeste, quando alguns deles ou todos existirem. Cada envio envolve no máximo $O(n^2)$ dados, como no caso A.

Desta forma, o algoritmo CGM para o caso B também roda em tempo local linear, $O(N^2/p)$, utiliza uma única rodada de comunicação, onde são enviados no máximo $O(n^2) = O(N^2/p)$ dados, e consome espaço $O(n^2) = O(N^2/p)$.

4.4 Resultados da Implementação

Na obtenção dos resultados práticos para este algoritmo, utilizamos uma matriz texto de ordem $N = 96$ e matrizes padrão de ordem $m = 4$ e $m = 24$. Estas matrizes eram formadas pelos símbolos a e b. Cada linha era constituída por estes símbolos alternadamente e duas linhas consecutivas não começavam com o mesmo símbolo. A posição na primeira linha e primeira coluna, de qualquer das matrizes, continha o símbolo a. Para estas matrizes existiram k-ocorrências do padrão em $N/2 - m$ posições das $N - m + 1$ primeiras linhas, com $k = 1$. Para estes dados não existiram ocorrências para qualquer outra escala $1 < k \leq \lfloor N/m \rfloor$. Com estes exemplos, temos a maior quantidade possível de dados trocados. Foram utilizados 1, 4 e 16 processadores da máquina *Parsytec PowerXplorer*.

Os resultados no gráfico da Figura 4.2 mostram os tempos em s. O comportamento do gráfico é muito parecido com os caso do capítulo anterior. O algoritmo rodando em um processador apresenta desempenho melhor para um padrão maior, uma vez que a quantidade de computação local acaba sendo menor. Os tempos caem mais acentuadamente quando passamos de 1 para 4 processadores.

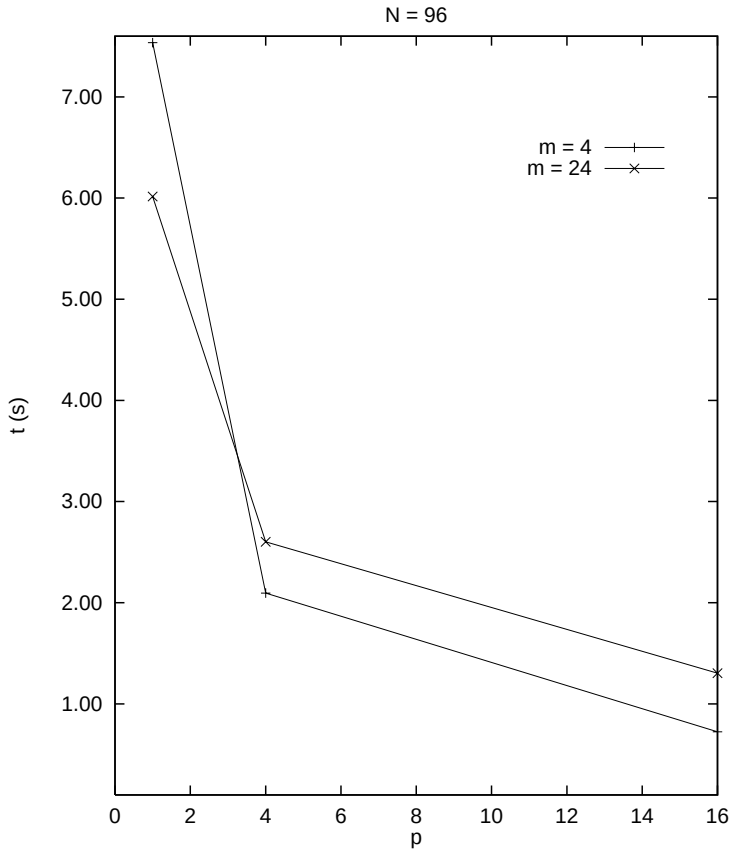

Figura 4.2: Tempo em s para $N = 96$, $m = 4$ e $m = 24$, utilizando 1, 4 e 16 processadores.

No gráfico da Figura 4.3 observamos um *speedup* bem significativo, próximo ao ótimo, para 4 processadores e um padrão de ordem $m = 4$. Este *speedup* se torna menor para 16 processadores. Para o padrão de ordem $m = 24$, obtemos *speedups* mais discretos, porém melhores que, por exemplo, os *speedups* da versão CGM para o algoritmo seqüencial sublinear para busca bidimensional sem escala. Uma das causas é a complexidade maior do algoritmo, principalmente por causa da árvore de sufixos que, neste caso, trabalha com uma cadeia bem maior, uma vez que todas as escalas do padrão estão nela inseridas.

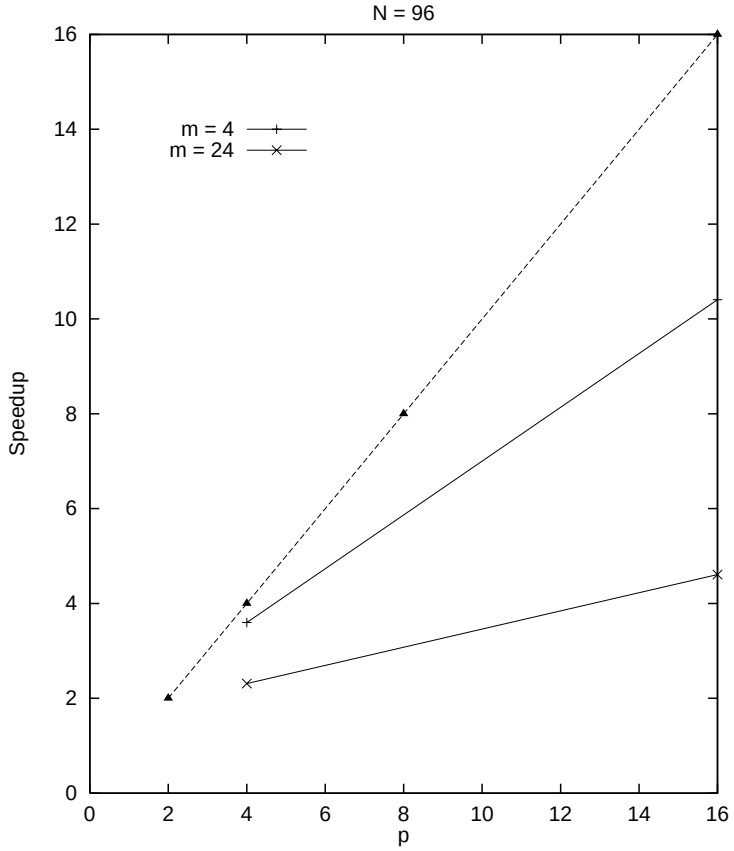

Figura 4.3: *Speedup* para 4 e 16 processadores com $N = 96$, $m = 4$ e $m = 24$.

No apêndice A encontram-se os valores dos tempos absolutos e dos *speedups*.

4.5 Notas

Neste capítulo, apresentamos inicialmente um algoritmo seqüencial para busca bidimensional de padrões com escala descrito por Amir *et al* [8]. Este algoritmo tem como entrada uma matriz texto de ordem N e uma matriz padrão de ordem m, com $m \leq N$, determinando em tempo $O(N^2)$ todas as k-ocorrências do padrão no texto, onde $1 \leq k \leq \lfloor N/m \rfloor$.

Baseados neste algoritmo, construímos um algoritmo CGM para o problema, utilizando p processadores. Este algoritmo é a principal contribuição desta tese. Ele tem tempo de computação local $O(N^2/p)$, consome memória também $O(N^2/p)$, utiliza uma rodada de comunicação onde são trocados no máximo $O(Nm/\sqrt{p})$ dados.

Com os dados experimentais obtidos com a implementação deste algoritmo, aferimos *speed-ups* bastante significativos. Para padrões pequenos e utilizando 4 processadores, o *speedup* foi quase ótimo. Para padrões maiores, o *speedup* foi mais discreto, porém melhor que os obtidos nas implementações dos algoritmos dos Capítulos 2 e 3.

Capítulo 5

Conclusões

Amir *et al* [8] apresentaram um algoritmo seqüencial para busca bidimensional de padrões com escala que obtém todas as ocorrências, para todas as escalas, em tempo seqüencial linear no tamanho da entrada. Este problema aparece em aplicações de busca em imagens, onde se permite uma escala ao padrão a ser buscado. Para descrever os algoritmos, os autores forneceram, também, algoritmos para os casos de busca unidimensional de padrões com escala e de busca bidimensional sem escala. Na busca bidimensional sem escala foram descritos dois algoritmos: um de tempo linear e um de tempo sublinear, sob algumas condições nos dados de entrada.

Utilizando as descrições seqüenciais, projetamos algoritmos paralelos no modelo CGM para estes problemas, utilizando p processadores. Para o caso unidimensional, considerando uma cadeia texto de comprimento N e uma cadeia padrão de comprimento m, os algoritmos CGM têm tempo de computação local $O(N/p)$, consumindo memória também $O(N/p)$ e utilizando uma rodada de comunicação, na qual são trocados no máximo $O(m)$ dados, onde $m \leq N/p$. Para o caso bidimensional, considerando como entrada uma matriz texto, quadrada, de ordem N e uma matriz padrão, também quadrada, de ordem m, os algoritmos CGM têm tempo de computação local $O(N^2/p)$, consumindo memória $O(N^2/p)$ e utilizando uma rodada de comunicação onde são trocados no máximo $O(Nm/\sqrt{p})$ dados, onde $m \leq N/\sqrt{p}$.

Os algoritmos CGM apresentados, neste trabalho, para os problemas de busca de padrões são inéditos, não sendo encontrados, na literatura, algoritmos em modelos de memória distribuída para estes problemas. Contribuímos, ainda, com o algoritmo CGM para o problema de mínimo intervalar [47, 46], que é utilizado no algoritmo CGM para o problema de busca bidimensional com escala.

No desenvolvimento destes algoritmos, consideramos regiões de fronteira que correspondem, no caso unidimensional, a prefixos e/ou sufixos do texto localmente armazenado e, no caso bidimensional, a submatrizes da matriz de texto localmente armazenada. Uma idéia inicial, e que funciona perfeitamente, é enviar, para os respectivos vizinhos, as regiões de fronteira e aplicar o algoritmo seqüencial nos dados estendidos (formados pelos dados locais mais as fronteiras recebidas). Esta solução impõe uma quantidade fixa de dados comunicados que é, na maioria das instâncias, muito maior que a quantidade de ocorrências. Nossos algoritmos enviam

para os vizinhos somente as posições que possam ser úteis na determinação das ocorrências. Esta preocupação com a quantidade de informações trocadas não aumenta excessivamente o tempo de computação local e, como o gargalo nestes casos é a comunicação, uma quantidade menor de dados trocados é preferível. Estas considerações são cruciais para a obtenção de algoritmos eficientes na teoria e na prática, quando implementados.

Estes algoritmos paralelos foram implementados na linguagem C, utilizando *interface* PVM e executados na máquina *Parsytec PowerXplorer*. Os dados experimentais obtidos mostraram um bom desempenho dos algoritmos para padrões pequenos e um desempenho mais discreto para padrões maiores. Isto é causado, principalmente, pela quantidade de dados trocados, uma vez que consideramos as instâncias de pior caso, onde a quantidade de dados trocados fosse máxima. Além disso, a comunicação na máquina *Parsytec PowerXplorer* é tipo *Ethernet* que acaba contribuindo também para uma demanda maior de tempo.

Como vimos, um algoritmo CGM para busca unidimensional sem escala pode ser construído de forma que a comunicação envolva apenas o envio de um inteiro. Esta idéia pode ser estendida ao caso bidimensional sem escala de forma que tenhamos uma quantidade mínima de dados a serem enviados (no mínimo um inteiro por coluna).

Para o caso bidimensional sem escala, podemos construir um algoritmo CGM híbrido que utilize nosso algoritmo ou a idéia descrita no parágrafo anterior de forma a minimizar a quantidade de dados comunicados. Neste algoritmo seria escolhida uma das duas formas para comunicação e manipulação dos dados, conforme a quantidade de dados a serem comunicados.

Obtivemos os dados experimentais apenas para o pior caso e mesmo assim acabamos obtendo resultados bastante animadores. Na prática, a quantidade de informações trocadas é bem menor que a considerada no caso médio o que deve levar a *speedups* melhores.

Para o caso bidimensional com escala, podemos utilizar algumas sugestões contidas no artigo original [8] de forma a otimizarmos a implementação. Não utilizamos estas sugestões em nossa implementação pois tínhamos uma quantidade pequena de memória física e se as acatássemos seria necessário reduzir o tamanho das matrizes de entrada.

Uma análise mais detalhada pode ser feita considerando a quantidade média de dados trocados.

Os problemas tratados neste trabalho foram motivados pelo problema de mínimo intervalar, para o qual já havíamos obtido um algoritmo CGM com *speedups* bastante satisfatórios. Deste problema ainda obtivemos um algoritmo, ainda em rascunho, para o problema de LCA.

As idéias contidas nos algoritmos, quanto a utilização de fronteiras, podem ser usadas em outras aplicações envolvendo vetores e matrizes que tenham a característica de dependência de informações em uma vizinhança.

Além de conseguirmos algoritmos inéditos para modelos de memória distribuída de granularidade grossa, a implementação dos algoritmos validou a utilização do modelo CGM. Os resultados experimentais obtidos foram muito bons, considerando que utilizamos uma máquina que não possui comunicação eficiente.

Apêndice A

Tabelas

As tabelas listadas a seguir apresentam os tempos absolutos e os *speedups* obtidos na parte experimental, executada na máquina *Parsytec PowerXplorer*.

Proc.	$m = 2^4$	$m = 2^{14}$
1	233221	239248
2	117476	239275
4	79504	208653
8	50264	189395
16	29758	163437

Tabela A.1: Tempos absolutos em μs para busca unidimensional de padrão sem escala em um texto de comprimento $N = 2^{18}$.

Proc.	$m = 2^4$	$m = 2^{14}$
2	1.9852651	0.99988716
4	2.9334499	1.146631
8	4.6399212	1.2632224
16	7.8372538	1.4638546

Tabela A.2: *Speedups* para busca unidimensional de padrão sem escala em um texto de comprimento $N = 2^{18}$.

Proc.	$m = 2^4$	$m = 2^{14}$
1	1519874	1546170
2	760621	963495
4	400678	650611
8	211623	472969
16	116178	371269

Tabela A.3: Tempos absolutos em μs para busca unidimensional de padrão com escala em um texto de comprimento $N = 2^{18}$.

Proc.	$m = 2^4$	$m = 2^{14}$
2	1.9982015	1.6047515
4	3.7932554	2.3764892
8	7.1819887	3.2690726
16	13.082288	4.1645545

Tabela A.4: *Speedups* para busca unidimensional de padrão com escala em um texto de comprimento $N = 2^{18}$.

Proc.	$m = 4$	$m = 24$
1	2711124	2558342
4	742431	1448536
16	226832	912686

Tabela A.5: Tempos absolutos em μs para busca bidimensional de padrão sem escala em um texto de ordem $N = 96$ (correspondente ao algoritmo seqüencial linear).

Proc.	$m = 4$	$m = 24$
4	3.6516848	1.766157
16	11.952123	2.8030911

Tabela A.6: *Speedups* para busca unidimensional de padrão sem escala em um texto de ordem $N = 96$ (correspondente ao algoritmo seqüencial linear).

Proc.	$m = 4$	$m = 24$
1	4148074	3430627
4	1103339	1946927
16	309242	1347917

Tabela A.7: Tempos absolutos em μs para busca bidimensional de padrão sem escala em um texto de ordem $N = 96$ (correspondente ao algoritmo seqüencial sublinear).

Proc.	$m = 4$	$m = 24$
4	3.7595644	1.7620727
16	13.413682	2.5451322

Tabela A.8: *Speedups* para busca unidimensional de padrão sem escala em um texto de ordem $N = 96$ (correspondente ao algoritmo seqüencial sublinear).

Proc.	$m = 4$	$m = 24$
1	7536214	6014806
4	2094936	2602478
16	724275	1304239

Tabela A.9: Tempos absolutos em μs para busca bidimensional de padrão com escala em um texto de ordem $N = 96$.

Proc.	$m = 4$	$m = 24$
4	3.5973481	2.3111842
16	10.405183	4.611736

Tabela A.10: *Speedups* para busca unidimensional de padrão com escala em um texto de ordem $N = 96$.

Referências Bibliográficas

[1] A. V. Aho, J. E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1975.

[2] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report TR 71/87, The Moise and Frida Eskenasy Inst. of Computer Science, Tel Aviv University, 1987.

[3] A. Amir, G. Benson, and M. Farach. Alphabet independent two dimensional matching. In *Proceedings of 24th Annual ACM Symposium on Theory of Computing*, ACM, pages 59–68, 1992.

[4] A. Amir, G. Benson, and M. Farach. Parallel two dimensional matching in logarithmic time. In *Proceedins of the 5th Annual Symposium on Parallel Algorithms and Architectures*, pages 79–85, 1993.

[5] A. Amir, G. Benson, and M. Farach. Optimal parallel two dimensional text searching on a CREW PRAM. *Information and Computation*, 144(1):1–17, 1998.

[6] A. Amir, A. Butman, and M. Lewenstein. Real scaled matching. *Information Processing Letters*, 70(4):185–190, 1999.

[7] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:347–365, 1988.

[8] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13:2–32, 1992.

[9] R. Baeza-Yates and M. Régnier. Fast algorithms for two-dimensional and multiple pattern matching. In *2nd Scandinavian Workshop on Algorithms Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 332–347, 1990.

[10] R. Baeza-Yates and M. Régnier. Fast two dimensional pattern matching. *Information Processing Letters*, 45(1):51–57, 1993.

[11] T. J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7:533–541, 1978.

[12] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14:344–370, 1993.

[13] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[14] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6:168–170, 1977.

[15] L. Boxer, R. Miller, and A. Rau-Chaplin. Scalable parallel algorithms for geometric pattern recognition. *Journal of Parallel and Distributed Computing*, 58(3):466–486, 1999.

[16] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming - ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400, 1997.

[17] L. Colussi. Fastest pattern matching in strings. *Journal of Algorithms*, 16:163–189, 1994.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[19] M. Crochemore, L. Gąsieniec, R. Hariharan, S. Muthukrishnan, and W. Rytter. A constant time optimal parallel algorithm for two-dimensional pattern matching. *SIAM J. Comput.*, 27(3):668–681, 1998.

[20] M. Crochemore, L. Gąsieniec, W. Plandowski, and W. Rytter. Two-dimensional pattern matching in linear time and small space. In *12th Annual Symposium on Theoretical Aspects of Computer Science*, volume 900 of *Lecture Notes in Computer Science*, pages 181–192, 1995.

[21] M. Crochemore, L. Gąsieniec, and W. Rytter. Two dimensional pattern matching by sampling. *Information Processing Letters*, 46(4):159–162, 1993.

[22] M. Crochemore and W. Rytter. On two dimensional pattern matching by optimal parallel algorithms. *Theoretical Computer Science*, 132:403–414, 1994.

[23] M. Crochemore and W. Rytter. On linear-time alphabet-independent 2-dimensional pattern matching. In *Proceedings of LATIN'95*, volume 911 of *Lecture Notes in Computer Science*, pages 220–229, 1995.

[24] F. Dehne. Guest editor's introduction. *Algorithmica*, 24(2):173–176, 1999.

[25] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar. A randomized parallel $3D$ convex hull algorithm for coarse grained multicomputers. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*, pages 27–33, 1995.

[26] F. Dehne, A. Fabri, and C. Kenyon. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 586–593, 1994.

[27] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proceedings of the ACM 9th Annual Computational Geometry*, pages 298–307, 1993.

[28] F. Dehne and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. In *Proceedings of the Second Asian Computing Science Conference (ASIAN'96)*, pages 1–10, 1996.

[29] J. Fan and K. Su. The design of efficient algorithms for two-dimensional pattern-matching. *IEEE Transactions on Knowledge and Data Engineering*, 7(2):318–327, 1995.

[30] P. Ferragina and R. Grossi. Optimal on-line search and sublinear time update in string search. *SIAM J. Comput.*, 27(3):713–736, 1998.

[31] P. Ferragina and F. Luccio. String search in coarse-grained parallel computers. *Algorithmica*, 24(2):177–194, 1999.

[32] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th ACM Symp. On Theory of Computing*, pages 135–143, 1984.

[33] Z. Galil. Optimal parallel algorithms for string matching. *Information and Control*, 67:144–167, 1985.

[34] Z. Galil and K. Park. Truly alphabet-independent two dimensional matching. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 247–256, 1992.

[35] R. Giancarlo. An index data structure for matrices with applications to fast two-dimensional pattern matching. In *3rd Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 337–348, 1993.

[36] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 17(1):128–142, 1988.

[37] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

[38] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[39] R. Karp and M. Rabin. Efficient randomized pattern matching algorithm. *IBM J. Res. Develop.*, 31:249–260, 1987.

[40] Z. Kedem, G. Landau, and K. Palem. Optimal parallel suffix-prefix matching algorithm and applications. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 388–398, 1989.

[41] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323–350, 1977.

[42] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[43] R. Lin and S. Olariu. A simple optimal parallel algorithm to solve the lowest common ancestor problem. In *Proceedings of the International Conference on Computing and Information - ICCI'91*, volume 497 of *Lecture Notes in Computer Science*, pages 455–461, 1991.

[44] T. Mathies. A fast parallel algorithm to determine edit distance. Technical Report CMU-CS-88-130, Cannergie-Mellon University, Pittsburgh, PA, 1988.

[45] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.

[46] H. Mongelli and S. W. Song. Parallel range minima for coarse grained multicomputers. *International Journal of Foundations of Computer Science*, 10(4):375–389, 1999.

[47] H. Mongelli and S. W. Song. A range minima parallel algorithm for coarse grained multicomputers. In *Workshop on Solving Irregularly Structured Problems in Parallel*, volume 1586 of *Lecture Notes in Computer Science*, pages 1075–1084, 1999.

[48] V. L. Ramachandran and U. Vishkin. Efficient parallel triconnectivity in logarithmic parallel time. In *Proceedings of the AWOC'88*, number 319 in Lecture Notes in Computer Science, pages 33–42, 1988.

[49] J. H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, 1993.

[50] E. M. Reingold, R. J. Urban, and D. Gries. KMP string matching revisited. *Information Processing Letters*, 64(5):217–223, 1997.

[51] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.

[52] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33:103–111, 1990.

[53] U. Vishkin. Optimal parallel pattern matching in strings. *Information and Control*, 67:91–113, 1985.

[54] J. Vuillemin. A unified look at data structures. *Comm. of the ACM*, 23:229–239, 1980.

[55] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th. IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.