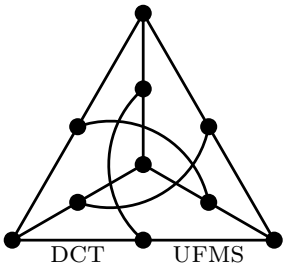
Algoritmos Paralelos Realísticos para a Maior Subsequência Comum

Claudia Regina Tinós Peviani

Dissertação de Mestrado

Orientação: Prof. Dr. Marco Aurélio Stefanes

Área de Concentração: Ciência da Computação



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
agosto de 2009

Algoritmos Paralelos Realísticos para a Maior Subsequência Comum

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, Departamento de Computação e Estatística da Fundação Universidade Federal de Mato Grosso do Sul.

Orientador: Prof. Dr. Marco Aurélio Stefanes

agosto de 2009

Agradecimentos

Ao Deus que sirvo e que é fiel a todo ser que nele crê, pela oportunidade de testar minha fé e sua fidelidade para comigo nessa caminhada.

Ao meu orientador, professor Dr. Marco Aurélio Stefanes, pela dedicação, paciência e confiança em mim depositadas.

A todos os professores do DCT, pelo apoio, pelos ensinamentos e pelas dicas. Não citarei nomes para não ser injusta com nenhum por esquecimento.

A todos os funcionários da Universidade Federal de Mato Grosso do Sul, em especial, a secretária do mestrado Beth pelas palavras de incentivo e motivação.

Aos meus colegas de Mestrado, pelos grupos de estudos antes das provas.

Ao meu amigo André Chastel Lima, pelas horas de estudos durante todo o período do Mestrado.

À toda minha família pelo carinho e amor, principalmente ao meu pai e minha mãe pelas palavras de perseverança.

Ao meu esposo Éder, por compreender minha ausência em muitos momentos da nossa vida e por cuidar dos nossos filhos nesses momentos com muita dedicação.

Aos meus filhos Carolina e Lucas, com quem aprendi o verdadeiro significado do amor incondicional, pela paciência, amor e compreensão.

Aos meus alunos pelas palavras de apoio e por ter compreendido minha ausência nas aulas.

A Deus sempre.

Resumo

Neste trabalho estudou-se o problema da Maior Subsequência Comum (*Longest Common Subsequence* - LCS) que deseja encontrar uma subsequência comum de comprimento máximo de duas sequências. O problema LCS pode ser resolvido em tempo sequencial quadrático usando a técnica de programação dinâmica. Este trabalho consiste em estudar os algoritmos paralelos baseados nos modelos realísticos para resolver o problema LCS. Inicialmente estudou-se um algoritmo sequencial, um algoritmo paralelo no modelo PRAM e um no modelo realístico que encontra a LCS de todas as subcadeias. Fez-se uma implementação paralela realística com $O(p)$ rodadas de comunicação usando p processadores. O resultado desta implementação não obteve *speedup* satisfatório, o que motivou o estudo mais teórico da paralelização do problema. Neste sentido, a partir do algoritmo PRAM descreveu-se uma solução paralela realística, usando estrutura de dados bem conhecidas como soma prefixa paralela e a técnica de *pointer jumping*.

Palavra-chave: LCS, Técnica de Programação Dinâmica, Algoritmos Paralelos.

Abstract

In this work, it was studied the problem of the Longest Common Subsequence - LCS that aims to finds a common subsequence of maximum lenght of two sequences. The LCS problem can be solved in quadratic sequential time using the dynamic programmimg technique. This work consists in studying the parallel algorithms based on the realistic models to solve the LCS problem. Initially they studied a sequential algorithm, a parallel algorithm PRAM model and one at the realistic model that finds the LCS of all substrings. It was done a realistic parallel implementation with $O(p)$ communication rounds using processors (p). The result of this implementation didn't get a satisfactory speedup that led to a more theoretical study of the problem parallelization. Thus, from the PRAM algorithm it was described a realistic parallel solution, using well-known data structure as parallel prefix sum and the pointer jumping technique.

Key words: LCS, Dynamic Programming Technique, Parallel Algorithms.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Um sistema de computação paralela consiste de um conjunto de processadores interligados que resolve um determinado problema de forma mais rápida que a computação sequencial que utiliza somente um processador. Sua grande motivação tem sido em resolver problemas com grande quantidade de informações, como é o caso: previsão de tempo, biologia computacional, processamento de imagens, inteligência artificial, modelagem e simulação de grandes sistemas, entre outros [Jáj92][Rei93].

Na computação paralela não existe um modelo único amplamente aceito. Os algoritmos paralelos são analisados pela complexidade de tempo e os recursos utilizados. Há algumas tentativas de se criar um modelo que sirva de paradigma para toda a área. Atualmente tem ganho aceitação os modelos chamados "Modelos Realísticos", cujo objetivo é produzir resultados eficientes, tanto do ponto de vista teórico quanto prático. Dentre estes modelos destacam-se o BSP (*Bulk Synchronous Parallel*) [Val90] e o CGM(*Coarse Grained Multicomputer*) [DFRC93].

Na computação paralela tem-se feito pouco estudo da técnica de programação dinâmica usando os modelos realísticos, a despeito do grande número de aplicações usando esta técnica. A técnica de programação dinâmica soluciona um problema de otimização através da combinação de soluções para seus subproblemas. Esses subproblemas não são independentes, isto é, compartilham subproblemas menores. Um algoritmo de programação dinâmica resolve cada subproblema uma vez só e então grava sua resposta em uma tabela, evitando assim o trabalho de recalcular a resposta toda vez que o subproblema é encontrado [CLRS02].

O problema da maior subsequência comum, ou LCS (do inglês Longest Common Subsequence), consiste em dadas duas sequências X e Y encontrar uma subsequência comum de X e Y de comprimento máximo. O problema LCS pode ser resolvido

na computação sequencial em tempo quadrático, para sequências de mesmo comprimento, usando-se a técnica de programação dinâmica [CLRS02].

Este problema possui várias aplicações como na biologia molecular, onde é utilizado para a comparação de sequências de DNA e no mapeamento de proteínas [SM97], na correção ortográfica e na comparação de arquivos.

1.1 Definições

Nesta seção definem-se dois conceitos fundamentais que são abordados ao longo deste trabalho.

Speedup mede o fator ganho obtido pelo algoritmo paralelo quando comparado ao melhor algoritmo sequencial para o problema.

$$S_p(n) = T^*(n)/T_p(n).$$

onde:

$T^*(n)$ - tempo do melhor algoritmo sequencial;

n - o tamanho da entrada;

$T_p(n)$ - tempo de um algoritmo paralelo com p processadores;

$S_p(n)$ - *Speedup* ou ganho.

É fácil verificar que o melhor *speedup* possível de ser alcançado é $S_p(n) \leq p$. Para obter um algoritmo paralelo com *speedup* ótimo, o algoritmo paralelo deveria produzir um $(S_p(n) = \Theta(p))$ sempre que $p = \Theta(T^*(n)/T_p(n))$.

Granulosidade Grossa é o termo usado quando o tamanho do problema é consideravelmente maior que o número de processadores, ou seja, $n/p \gg p$.

Cadeia é uma sequência finita de símbolos tomados de um determinado alfabeto finito.

Subcadeia é uma sequência equivalente a um "trecho" de símbolos contíguos de uma cadeia.

Sequência idem a cadeia, combinação de símbolos pertencentes a um alfabeto finito .

Subsequência é uma sequência de símbolos selecionados de uma sequência que preserva a ordem em que estes estão, porém os símbolos presentes numa subsequência não precisam estar contínuos na sequência original.

1.2 Objetivo e Descrição do Trabalho

Este trabalho consiste em estudar os algoritmos paralelos para resolver o problema LCS baseados nos modelos realísticos. Inicialmente, será estudado o algoritmo sequencial existente e os algoritmos paralelos nos modelos PRAM e realísticos. É descrita uma implementação de um algoritmo paralelo para verificar a eficiência dos mesmos.

A técnica de programação dinâmica apresenta-se como uma ferramenta de auxílio no desenvolvimento de algoritmos eficientes em várias áreas. No campo da computação paralela, pouco tem se avançado no sentido de que os algoritmos sequenciais que resolvem problemas usando esta técnica sejam resolvidos eficientemente também em máquina paralelas. Assim sendo, tem-se como objetivo também conhecer mais profundamente as características desta técnica e analisar as possibilidades de ampliar seu uso em algoritmos paralelos usando os modelos realísticos.

A seguir é descrito como este trabalho está estruturado. O trabalho está dividido em oito capítulos. O Capítulo 1 contém esta introdução. No Capítulo 2 apresenta-se os modelos de computação paralela presentes na literatura que serão muito úteis para o desenvolvimento do trabalho. Primeiramente é apresentada uma classificação dos modelos de computação paralela. Ainda no Capítulo 2 serão apresentados o Modelo de Memória Compartilhada, o Modelo de Memória Distribuída e, por fim, os Modelos Realísticos. No Capítulo 3, apresenta-se um algoritmo sequencial para o problema LCS, utilizando a técnica de programação dinâmica. No Capítulo 4, apresenta-se uma implementação da LCS no modelo CGM, é apresentado o ambiente computacional e os resultados dos testes. Um algoritmo paralelo no modelo CGM para resolver o problema ALCS é apresentado no Capítulo 5, sendo resolvido esse problema é simples encontrar a LCS. Um algoritmo no modelo PRAM é mostrado no Capítulo 6 e no Capítulo 7 é mostrado essa mesma ideia do algoritmo apresentado

no Capítulo 6 mas no modelo CGM. Finaliza-se este trabalho com o capítulo onde são apresentadas as considerações finais.

Capítulo 2

Modelos de Computação Paralela

Neste capítulo, apresentam-se alguns modelos de computação paralela. Inicialmente será visto como os sistemas de computação paralela podem ser classificados de acordo com suas características de arquitetura e modos de operação, além do sincronismo das operações. Na seção 2.2 são apresentados os modelos de computação paralela.

2.1 Sistema de Computação Paralela

Os computadores paralelos podem ser classificados através de suas características de arquitetura e modos de operação. Incluem a interconexão entre processadores e seus respectivos esquemas de comunicação, controle e sincronismo das operações.

Uma maneira de classificar arquiteturas de computadores é pela forma como o fluxo de instruções e os dados se apresentam. Essa classificação é conhecida por taxonomia de Flynn [Fly72], onde destacam-se os sistemas SIMD (*Single Instruction Multiple Data*) e o MIMD (*Multiple Instruction Multiple Data*).

Computador paralelo SIMD todos os processadores executam sincronamente a mesma instrução, usando diferentes informações nesta instrução.

Computador paralelo MIMD os processadores podem executar diferentes programas simultaneamente, cada processador armazena em sua própria memória local o programa. São computadores assíncrono e síncrono.

De acordo com o sincronismo das operações têm-se:

Computador paralelo síncrono quando todos os processadores executam uma instrução (não precisam ser idênticas) sob o controle de um relógio comum.

Computador paralelo assíncrono quando é necessário incluir pontos de sincronização entre os processadores, ou seja, não há um relógio comum para todos os processadores.

2.2 Modelos de Computação Paralela

Uma variedade de modelos foi proposta para a computação paralela. Não há um modelo que seja amplamente utilizado, como no modelo sequencial, para projeto e análise de algoritmos paralelos. Apresentam-se três modelos que são bastante utilizados no desenvolvimento e análise de algoritmos paralelos. Os modelos são: Modelo de Memória Compartilhada, Modelo de Memória Distribuída e Modelos Realísticos.

2.2.1 Modelo de Memória Compartilhada

O modelo de memória compartilhada consiste de um número de processadores, cada um com uma pequena memória local, que executa seu próprio programa local. A comunicação é feita pela troca de informações através da memória global.

Modelo PRAM (Máquina Paralela com Memória de Acesso Aleatório) é uma coleção de processadores síncronos executando em paralelo e se comunicando via memória compartilhada. Modelo PRAM é do tipo MIMD, ou seja, quando cada processador pode executar uma instrução diferente daquela que está sendo processada pelo outro processador e utilizando uma mesma unidade de tempo.

Variações do Modelo PRAM em relação ao acesso concorrente de uma mesma posição de memória global são:

PRAM EREW PRAM de Leitura e Escritas Exclusivas, não permite acesso concorrente.

PRAM CREW PRAM de Leitura Concorrente e Escrita Exclusiva, permite leitura, mas não permite alteração.

PRAM CRCW PRAM com Leitura e Escrita Concorrente, neste caso é preciso criar critérios para a escrita concorrente. Podem ser adotadas várias políticas para decidir qual o processador irá efetuar a escrita:

PRAM CRCW Comum permite a escrita concorrente somente quando for o mesmo valor para todos os processadores.

PRAM CRCW Arbitrário um processador arbitrário terá sucesso na escrita.

PRAM CRCW Prioritário os processadores têm prioridade e aquele com prioridade maior é que terá direito a escrita.

Dessas variações de modelos PRAM, o que tem maior poder computacional é o PRAM CRCW Prioritário e o modelo PRAM EREW é o mais fraco.

2.2.2 Modelo de Memória Distribuída

Conhecido como modelo de redes, é um conjunto de processadores onde a memória está distribuída entre os processadores e não há a memória global disponível.

Este modelo por ser representado por um grafo $G = (V, E)$ com $i \in V$ representando um processador e cada $(i, j) \in E$ representando uma linha de comunicação de mão dupla entre os processadores i e j. O modelo de memória distribuída pode operar nos modos síncrono ou assíncrono. A comunicação é feita via troca de mensagem, somente quando necessário. Existem algumas topologias representativas de redes de interconexão:

Vetor Linear e o Anel consiste de p processadores $P_1, P_2, \ldots, P_p$ conectados em uma lista linear e o anel é a lista linear onde P_1 e P_p estão ligados diretamente.

Malha é a versão bidimensional do vetor linear.

Árvore Binária consiste de $p = 2^d - 1$ processadores onde cada P_i está conectado aos processadores P_{2i} e P_{2i+1} chamados de filhos de P_i e ao processador $P_{\lfloor i/2 \rfloor}$ chamado de pai de P_i. O grau máximo de uma árvore binária é três.

Hipercubo consiste de $p = 2^d$ processadores interligados em um cubo d-dimensional. Possui uma estrutura recursiva.

2.2.3 Modelos Realísticos

Os modelos realísticos surgem da busca de um modelo adequado com características intrínsecas à computação paralela. Esse modelo pode ser visualizado em um ambiente de inovações tecnológicas aceleradas, apesar de muito abstrato. Esse modelo deveria balancear simplicidade com precisão e abstração com praticidade fornecendo uma visão sobre computação paralela [MMT95].

Os principais modelos realísticos são:

Modelo BSP (Modelo Paralelo Síncrono Volumoso) proposto por Valiant[Val90], foi um dos primeiros modelos a considerar os custos de comunicação como característica de um modelo de computação paralela. O objetivo do modelo é servir de ponte entre os algoritmos e o hardware. Uma máquina BSP é formada por p processadores com memória local que se comunicam através de algum meio de interconexão, gerenciados por um roteador e com uma sincronização periódica global. Um algoritmo BSP consiste em uma sequência de superpassos. Em cada superpasso os processadores operam independentemente realizando computações locais e comunicações globais através de envio e recebimento de mensagens.

Modelo CGM (Multicomputador com Granulosidade Grossa) que foi proposto por Dehne et al. [DFRC93] e é derivado do BSP. O CGM é definido em apenas dois parâmetros:

1. n: tamanho do problema;
2. p: número de processadores $P_1, P_2, \ldots, P_p$, cada um com uma memória local de tamanho $O(n/p)$.

Um algoritmo CGM consiste de fases de processamento local alternadas com rodadas de comunicação global, onde em cada rodada a quantidade de dados trocados por cada processador deve ser $O(n/p)$. O objetivo de um algoritmo CGM é minimizar o número de superpassos e a quantidade de computação local.

Modelo LogP proposto por [CKP+93], esse modelo incorpora mais atributos das máquinas existentes. Parâmetros do modelo:

- L: limite superior de latência;
- o: *overhead*, tempo que o processador fica parado;
- g: *gap*, intervalo de tempo mínimo entre a transmissão ou recepção de mensagens;
- P: número de processadores e módulos de memória.

Este modelo é semelhante ao BSP, mas não requer sincronizações em barreira. Isto permite melhor sintonia entre algoritmo e máquina, sobrepondo etapas de comunicação e processamento interno em diferentes processadores.

Capítulo 3

Algoritmo Sequencial para o problema LCS

Este capítulo, aborda o problema da Maior Subsequência Comum do inglês *Longest Common Subsequence*, com a sigla LCS. Na primeira seção é apresentada a técnica de programação dinâmica que será usada para resolver o problema LCS. Na última seção é apresentado um algoritmo sequencial que resolve o problema LCS.

Dada uma sequência $X = \langle x_1, x_2, \ldots, x_m \rangle$, outra sequência $Z = \langle z_1, z_2, \ldots, z_k \rangle$ é uma subsequência de X se existe uma sequência estritamente crescente $\langle i_1, i_2, \ldots, i_k \rangle$ de índices de X tais que, para todo $j = 1, 2, \ldots, k$, tem-se $x_{i,j} = z_j$ [CLRS02]. A sequência Z é uma subsequência comum de X e Y, se Z é uma subsequência de X e de Y ao mesmo tempo.

O problema LCS consiste em dadas duas sequência X e Y encontrar a subsequência comum de comprimento (números de símbolos) máximo entre as sequências.

Para resolver o problema LCS utiliza-se a técnica de programação dinâmica, este problema possui uma subestrutura ótima e é resolvido de forma eficiente e elegante.

3.1 Técnica de Programação Dinânica

A programação dinâmica, como o método de divisão e conquista, resolve problemas combinando as soluções para subproblemas. Os algoritmos de dividir e conquistar particionam o problema em subproblemas independentes, resolvem os subproblemas

recursivamente, e então combinam suas soluções para resolver o problema original. Por outro lado, a programação dinâmica é aplicada quando os subproblemas não são independentes, isto é, quando os subproblemas compartilham subproblemas menores. Um algoritmo de programação dinâmica resolve cada subproblema uma vez só e então grava sua resposta em uma tabela, evitando assim o trabalho de recalcular a resposta toda vez que o subproblema é encontrado [CLRS02].

A programação dinâmica é uma técnica para solução de problemas de otimização combinatória. Para tais problemas, podem haver muitas soluções possíveis, onde cada solução tem um valor e deseja-se encontrar uma solução com um valor ótimo (mínimo ou máximo). Esta solução encontrada é chamada de *uma* solução ótima, em vez de *a* solução ótima, pois podem haver várias soluções que alcancem o valor ótimo.

A técnica de programação dinâmica é uma ferramenta de auxílio para solucionar o problema LCS. Em geral, quando um problema de otimização possui subestrutura ótima e sobreposição de subproblemas, a técnica de programação dinâmica pode ser utilizada para solucionar tal problema. Um problema possui subestrutura ótima se uma solução ótima para o problema contém soluções ótimas para os subproblemas. Além disso, quando um procedimento recursivo revisita o mesmo problema diversas vezes, referencia a ele como o problema de otimização com sobreposição de subproblemas.

Uma vez que o comprimento destas sequências a serem analisadas pode ser enorme, é notória a importância da computação paralela como ferramenta de apoio a estes estudos. No entanto, existem poucos algoritmos paralelos para solucionar este problema com comprovada eficiência prática. Pretende-se com este trabalho contribuir para encontrar soluções paralelas eficientes tanto do ponto de vista teórico quanto prático.

3.2 Algoritmos Sequenciais

Uma solução eficiente do problema LCS faz uso de uma propriedade de subestrutura ótima existente no problema, como mostra o Teorema 3.2.1 a seguir. Dada uma sequência $X = \langle x_1, x_2, \ldots, x_m \rangle$, define-se o i-ésimo prefixo de X, para $i = 0, 1, \ldots, m$, como $X_i = \langle x_1, x_2, \ldots, x_i \rangle$.

Teorema 3.2.1 (Subestrutura ótima de uma LCS) *Sejam as sequências* $X = \langle x_1, x_2, \ldots, x_m \rangle$ *e* $Y = \langle y_1, y_2, \ldots, y_n \rangle$ *e seja* $Z = \langle z_1, z_2, \ldots, z_l \rangle$ *qualquer LCS de* X *e* Y*. Então,*

1. Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma LCS de X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma LCS de X_{m-1} e Y.
3. Se $x_m \neq y_n$, então $z_k \neq y_n$ implica que Z é uma LCS de X e Y_{n-1}.

A caracterização do Teorema 3.2.1 mostra que uma LCS de duas sequências contém dentro dela uma LCS de prefixos das duas sequências. Desse modo, o problema LCS tem uma propriedade de subestrutura ótima. A partir deste teorema pode-se obter uma solução recursiva para o problema. Esta solução possui a característica de fazer uso de subproblemas superpostos. Pode-se encontrar uma descrição mais completa e a demonstração desse teorema em [CLRS02].

3.2.1 Calcular o comprimento de uma LCS

Tendo em vista que existem apenas $\Theta(mn)$ subproblemas distintos, pode-se, através da programação dinâmica, calcular as soluções de baixo para cima.

O algoritmo Comprimento-LCS toma duas sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ como entradas. Ele armazena os valores $c[i, j]$ em uma tabela $c[0..m, 0..n]$ cujas entradas são calculadas em ordem de linha principal. Isto é, a primeira linha de c é preenchida da esquerda para a direita, depois a segunda linha e assim por diante. Ele também mantém a tabela $b[1..m, 1..n]$ para simplificar a construção de uma solução ótima. Intuitivamente, $b[i, j]$ aponta para a entrada da tabela correspondente à solução ótima do subproblema escolhida ao se calcular $c[i, j]$. O algoritmo devolve as tabelas b e c. O valor $c[m, n]$ contém o comprimento de uma LCS de X e Y.

Algoritmo 3.2.1 *Comprimento-LCS*

Entrada: Duas sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$
Saída: As tabelas b e c; $c[m, n]$ com o comprimento de uma LCS de X e Y.
1. $m \leftarrow$ *comprimento* $[X]$
2. $n \leftarrow$ *comprimento* $[Y]$
3. **Para** $i \leftarrow 1$ **até** m
4. **faça** $c[i, 0] \leftarrow 0$
5. **Para** $j \leftarrow 0$ **até** n
6. **faça** $c[0, j] \leftarrow 0$
7. **Para** $i \leftarrow 1$ **até** m
8. **faça Para** $j \leftarrow 1$ **até** n

9. **faça Se** $x_i = y_j$
10. **então** $c[i,j] \leftarrow c[i-1,j-1]+1$
11. $b[i,j] \leftarrow$ " $\nwarrow$ "
12. **senão Se** $c[i-1,j] \geq c[i,j-1]$
13. **então** $c[i,j] \leftarrow c[i-1,j]$
14. $b[i,j] \leftarrow$ " $\uparrow$ "
15. **senão** $c[i,j] \leftarrow c[i,j-1]$
16. $b[i,j] \leftarrow$ " $\leftarrow$ "
17. **devolva** c e b

A tabela 3.1 mostra as tabelas superpostas produzidas pelo algoritmo Comprimento-LCS para as sequências $X = \langle a,b,c,b,b,a,b,c,a,b,c,a\rangle$ e $Y = \langle c,b,a,c,b,a,a,b,a,b,a,c,b,a,a\rangle$. O tempo de execução do algoritmo é $O(mn)$, pois cada entrada da tabela demora tempo $O(1)$ para ser calculada.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i		y_j	***c***	***b***	***a***	***c***	***b***	***a***	***a***	***b***	***a***	***b***	***a***	***c***	***b***	***a***	***a***
0	x_i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	***a***	0	↑0	↑0	↖1	←1	←1	↖1	↖1	←1	↖1	←1	↖1	←1	←1	↖1	↖1
2	***b***	0	↑0	↖1	↑1	↑1	↖2	←2	←2	↖2	←2	↖2	←2	←2	↖2	←2	←2
3	***c***	0	↖1	↑1	↑1	↖2	↑2	↑2	↑2	↑2	↑2	↑2	↑2	↖3	←3	←3	←3
4	***b***	0	↑1	↖2	←2	←2	↖3	←3	←3	↖3	←3	↖3	←3	←3	↖4	←4	←4
5	***b***	0	↑1	↖2	←2	←2	↖3	←3	←3	↖4	←4	↖4	←4	←4	↖4	←4	←4
6	***a***	0	↑1	↑2	↖3	←3	←3	↖4	↖4	←4	↖5	←5	↖5	←5	←5	↖5	↖5
7	***b***	0	↑1	↖2	↑3	←3	↖4	←4	←4	↖5	←5	↖6	←6	←6	↖6	←6	←6
8	***c***	0	↖1	↑2	↑3	↖4	←4	←4	←4	↑5	←5	↑6	←6	↖7	←7	←7	←7
9	***a***	0	↑1	↑2	↖3	↑4	↑4	↖5	↖5	←5	↖6	←6	↖7	←7	←7	↖8	↖8
10	***b***	0	↑1	↖2	↑3	↑4	↖5	↑5	↑5	↖6	↑6	↖7	↑7	↑7	↖8	↑8	↑8
11	***c***	0	↖1	↑2	↑3	↖4	↑5	↑5	↑5	↑6	↑6	↑7	↑7	↖8	↑8	↑8	↑8
12	***a***	0	↑1	↑2	↖3	↑4	↑5	↖6	↖6	↑6	↖7	↑7	↖8	↑8	↑8	↖9	↖9

Tabela 3.1: As tabelas c e b calculadas por Comprimento-LCS sobre as sequências $X = \langle a,b,c,b,b,a,b,c,a,b,c,a\rangle$ e $Y = \langle c,b,a,c,b,a,a,b,a,b,a,c,b,a,a\rangle$

A tabela b devolvida por Comprimento-LCS pode ser usada para construir facilmente uma LCS de X e Y em tempo linear. Observe a tabela 3.1. Inicia-se na posição $b(m,n)$ e percorre a tabela seguindo as setas. Sempre que encontrar a seta " $\nwarrow$ " pode-se identificar onde há casamento, $x_i = y_j$, e imprimi-lo.

Capítulo 4

Uma Implementação do problema LCS no CGM

Neste capítulo, são apresentados os resultados obtidos através da implementação de um algoritmo simples para resolver o problema LCS no modelo CGM. Durante o decorrer deste capítulo será mostrada a descrição do algoritmo, o ambiente computacional e como foram gerados os dados de entrada juntamente com os resultados obtidos nos testes efetuados.

4.1 Descrição do Algoritmo

O algoritmo *LCS-CGM* toma duas sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ como entradas. A sequência X é distribuída uniformente entre os processadores e a sequência Y é enviada inteira para os processadores. É executado sequencialmente o algoritmo LCS em p processadores sobre as entradas recebidas. O algoritmo armazena os valores $c[i, j]$ em uma tabela $c[0..m, 0..n]$ cujas entradas são calculadas em ordem de linha principal. Isto é, a primeira linha de c é preenchida da esquerda para a direita, depois a segunda linha e assim por diante. O processador p_{k-1}, sendo $k = 1, \ldots, p$, envia para o processador p_k sua última linha da tabela c. O processador p_k recebe essa linha do processador p_{k-1} e atribui à linha 0 de sua tabela. Localmente, calcula a LCS sequencial, esse procedimento é executado k vezes. Veja abaixo a descrição do algoritmo *LCS-CGM*.

Algoritmo 4.1.1 *LCS-CGM*

Entrada: Duas sequências $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$
Saída: $c[m, n]$ com o comprimento da LCS de X e Y.
1. Dividir a sequência X entre os p processadores. Cada processador p_k, sendo $k = 0, \ldots, p-1$, recebe $\frac{m}{p}$ caracteres de X.
2. Enviar a sequência Y para os p processadores. Cada processador p_k recebe n caracteres de Y.
3. O processador p_0 executa localmente o Comprimento-LCS.
4. **Se** $p_{k=0}$ **então**
5. Executa Comprimento-LCS
6. **senão**
7. **Para** $i \leftarrow 0$ **até** $\frac{m}{p}$ **faça**
8. $c[i, 0] \leftarrow 0$
9. **Para** $j \leftarrow 0$ **até** n **faça**
10. $c[0, j] \leftarrow 0$
11. Cada processador p_k, sendo $k = 1, \ldots, p$, recebe a linha $\frac{m}{p}$ do processador p_{k-1} e executa Comprimento-LCS.
12. **Para** $i = 1$ **até** k **faça**
13. **Se** $p_{k=i-1}$ **então**
14. Envia $c[\frac{m}{p}, n]$ para p_k
15. **Se** $p_{k=i}$ **então**
14. Recebe $c[\frac{m}{p}, n]$ de p_{k-1}
15. Executa Comprimento-LCS
16. O último processador devolve o tamanho da LCS.

4.2 Ambiente Computacional

O algoritmo *LCS-CGM* foi implementado utilizando a linguagem C/C++ com a biblioteca mpi.h. O algoritmo foi executado em um Cluster. O Cluster é composto por 8 nós: uma máquina AMD Athlon(tm) 1800+, com 1GB de memória RAM; três máquinas Pentium IV 2.66GHz, com 512MB de memória RAM; quatro máquinas AMD Athlon(tm) 1.66GHz, com 480MB de memória RAM. Os nós estão conectados por um switch fast-Ethernet de 1Gb. Cada nó executa o sistema operacional Linux Fedora 6 com g++ 4.0 e MPI/LAM 7.1.2.

4.3 Dados de Entrada e Resultados

Para testar a implementação do algoritmo *LCS-CGM* utilizou-se como dados de entrada sequências geradas a partir de uma implementação simples. Dado o tamanho que se deseja gerar a sequência, o código gera aleatoriamente através do alfabeto finito {A, T, C, G} a sequência desejada e no tamanho desejado.

Foram geradas 6 sequências. Duas sequências de tamanho 1024, duas de tamanho 8192, uma de tamanho 16384 e uma de tamanho 32768. A escolha por esses tamanhos foi para facilitar a distribuição entre os processadores. Simplesmente utilizou-se 2^{10}, 2^{13}, 2^{14} e 2^{15} para chegar aos tamanhos mencionados anteriormente.

Na Tabela 4.1 observa-se a combinação que foi usada para gerar os resultados apresentados na Tabela 4.2. A sequência de tamanho 1024 foi executada com as sequências de tamanho 1024 que corresponde a letra A, 8192 que corresponde a letra B, 16384 que corresponde a letra C e 32768 que corresponde a letra D. A sequência de tamanho 8192 foi executada com as sequências de tamanho 8192 que corresponde a letra E, 16384 que corresponde a letra F e 32768 que corresponde a letra G. Não foi possível executar a sequência de tamanho 16384 com a sequência de tamanho 32768, pois o tamanho da tabela é inviável à capacidade de memória RAM do Cluster utilizado.

	1024	8192	16384	32768
1024	A	B	C	D
8192	-	E	F	G

Tabela 4.1: Tamanho das sequências.

Analisando o resultado apresentado na Tabela 4.2 pode-se notar que o LCS sequencial obteve melhor tempo do que *LCS-CGM* e não tem perspectiva de melhora. Observe o algoritmo *LCS-CGM*, são feitas $p-1$ rodadas de comunicação tranferindo a última linha dos $p-1$ processadores. Cada processador executa sequencialmente o LCS após receber do processador anterior a última linha da tabela, não há processamento paralelo, somente é usado o paralelismo na distribuição da sequência X uniformente. Quanto maior o número de processadores maior será o custo de comunicação.

Todos os tempos apresentados na Tabela 4.2 estão em segundos e não incluem o tempo de leitura dos dados de entrada e impressão da saída. O tempo de execução deste algoritmo é de fácil análise: em cada rodada de processamento o algoritmo leva tempo e ocupa espaço local $O(\frac{m}{p}n)$, sendo $p-1$ rodadas de computação, com $O(n)$ dados transferidos em cada rodada.

p	A	B	C	D	E	F	G
1	0.019	0.190	0.445	0.896	1.648	3.740	7.312
2	0.027	0.182	0.399	0.810	1.496	3.281	6.537
4	0.023	0.236	0.524	1.054	1.934	4.271	8.010
8	0.024	0.217	0.472	0.948	1.749	3.831	7.601

Tabela 4.2: Resultado da implementação do algoritmo *LCS-CGM*.

Estes resultados nos motivaram a buscar soluções mais elaboradas para o problema LCS, que reduz substancialmente o número de rodadas de comunicação.

Capítulo 5

Algoritmo Paralelo CGM para o problema ALCS

Neste capítulo, é abordado o problema da Maior Subsequência Comum e todas as Subcadeias, do inglês *All-substrings Longest Common Subsequence*-ALCS. Este algoritmo é a base para resolver o problema LCS. Primeiramente apresentam-se algumas definições necessárias para o problema ALCS. Em seguida define-se o problema ALCS e tem-se uma ideia do funcionamento do algoritmo sequencial e paralelo.

5.1 Definições Preliminares

Como o trabalho envolve estudar um algoritmo paralelo realístico para o problema da Maior Subsequência Comum, fazem-se necessárias algumas definições, como por exemplo diferenciar subcadeia e subsequência. Para isso utiliza-se um exemplo. Dada a sequência X composta por $\langle a, b, c, b, b, a, b, c, a, b, c, a\rangle$, onde o alfabeto finito $\{a, b, c\}$ é o conjunto utilizado.

Uma **subcadeia** de uma sequência X é um prefixo de algum sufixo de X ou sufixo de algum prefixo de X. Pode-se dizer que uma subcadeia de X é uma sequência equivalente a um "trecho" de símbolos contíguos de X. A subcadeia é indicada pela letra maiúscula correspondente à sequência original e índices que indicam a posição inicial (subscrito) e final (sobrescrito) da sequência. Se X é a sequência do exemplo anterior, então X_5^8 é a subcadeia $\langle b, a, b, c\rangle$.

Agora uma **subsequência** de X é uma sequência de símbolos selecionados de

X que preserva a ordem em que estes estão em X. Note que os símbolos presentes numa subsequência não precisam estar contínuos na sequência original, ao contrário do que ocorre na definição de subcadeias. Tem-se, como exemplos de subsequências de X, $\langle a, c, b, b, a, b, c, a, a\rangle$, $\langle a, c, b, b, a, b, a, c, a\rangle$ e $\langle a, c, b, a, b, a, b, a\rangle$.

Outra definição é o **Grafo Dirigido Acíclico do tipo Grade** conhecido também pela sigla **GDAG**. Considere duas sequências X e Y de comprimento m e n respectivamente. Um **GDAG** correspondente tem vértices que estão dispostos em um arranjo $(m+1) \times (n+1)$. As linhas e colunas são numeradas a partir de 0, o vértice da linha i e coluna j, $G(i, j)$ representa o valor do alinhamento dos prefixos X_1^i e Y_1^j.

O valor do alinhamento dos caracteres x_i e y_j é dado por uma função σ tal que $\sigma(x_i, y_j)$ para o problema LCS pode ser 1 para $x_i = y_j$ e 0 para $x_i \neq y_j$.

Os arcos do GDAG são assim definidos:

- **arcos diagonais**: para $1 \leq i \leq m$ e $1 \leq j \leq n$ há um arco de $G(i-1, j-1)$ para $G(i, j)$, de peso $\sigma(x_i, y_j)$.
- **arcos verticais**: para $0 \leq i \leq m$ e $1 \leq j \leq n$ há um arco de $G(i, j-1)$ para $G(i, j)$, de peso $\sigma(x_i, -)$.
- **arcos horizontais**: para $1 \leq i \leq m$ e $0 \leq j \leq n$ há um arco de $G(i-1, j)$ para $G(i, j)$, de peso $\sigma(-, y_j)$.

Um exemplo de GDAG é apresentado na Figura 5.1.

O problema ALCS é modelado por um GDAG. Para resolver este problema é preciso determinar o caminho de maior valor (maior soma dos pesos dos arcos percorridos) entre $G(0, 0)$ e $G(m, n)$. O GDAG pode ser adaptado para resolver o problema LCS, porém antes precisa-se entender como ele resolve o problema ALCS.

5.2 Problema ALCS

O problema ALCS consiste em dadas as sequências X e Y, encontrar a maior subsequência comum entre a sequência X e todas as possíveis subcadeias Y_i^j de Y, $1 \leq i \leq j \leq n$ onde n é o comprimento da sequência Y.

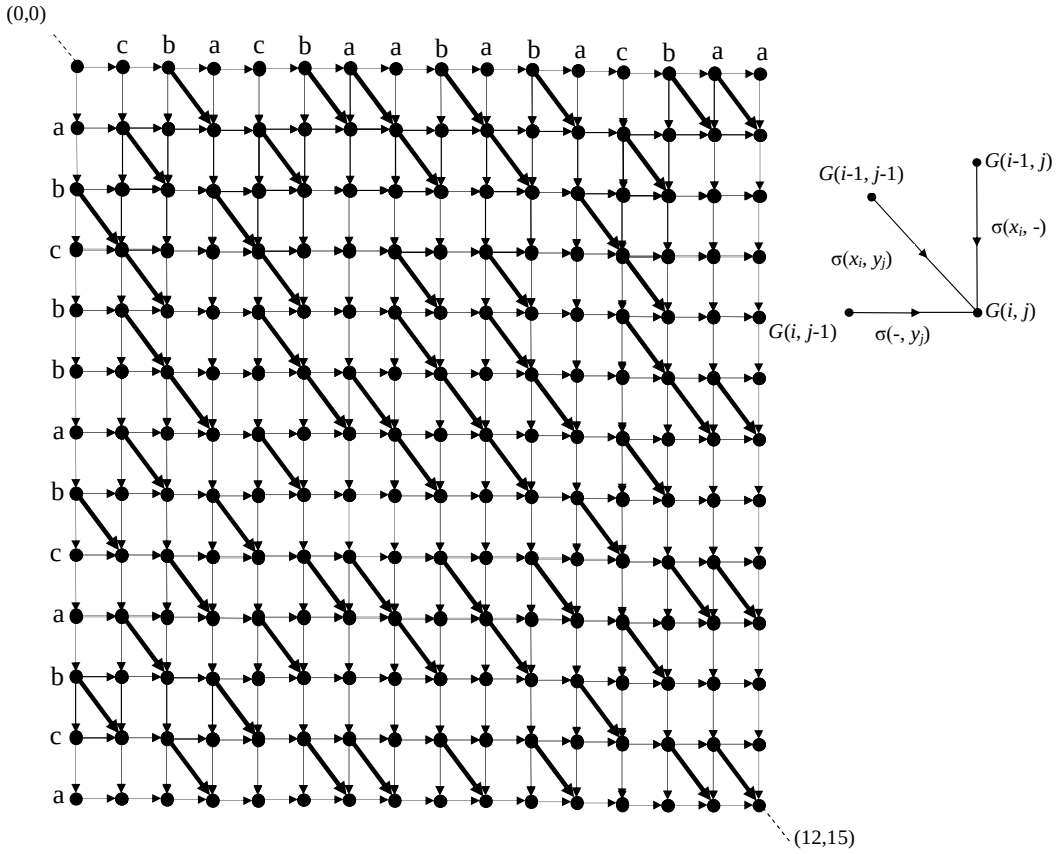


Figura 5.1: GDAG para o Problema ALCS entre as sequências $X = \langle a, b, c, b, b, a, b, c, a, b, c, a\rangle$ e $Y = \langle c, b, a, c, b, a, a, b, a, b, a, c, b, a, a\rangle$.

Considere o GDAG G que corresponde ao ALCS das sequências X e Y de comprimento m e n, respectivamente. Este GDAG possui $m + 1$ linhas e $n + 1$ colunas de vértices. Todos os arcos verticais e horizontais têm peso 0, arcos diagonais têm peso 1 se $x_i = y_j$. Se $x_i \neq y_j$, este arco tem peso 0, podendo ser desprezado.

Os vértices da linha superior (topo) de G são denotados por $T_G(i)$ e os vértices da linha inferior (fundo) de G por $F_G(i)$, $0 \leq i \leq n$. O $C_G(i, j)$ representa o comprimento da maior subsequência comum entre a sequência X e a subcadeia Y_{i+1}^{j}. A Tabela 5.1 mostra os valores de C_G para o GDAG da Figura 5.1. A solução para o problema ALCS pode ser escrita em uma matriz D_G, onde $D_G(i, 0) = i$, $0 \leq i < n$, e para $1 \leq k \leq m$, $D_G(i, k)$ indica o valor j tal que $C_G(i, j) = k$ e $C_G(i, j - 1) = k - 1$. Se não houver tal valor $D_G(i, k) = \infty$. A linha i da matriz D_G é denominada D_G^i. Observe a Tabela 5.2 que apresenta os valores de D_G obtidos da

Tabela 5.1.

	j															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$C_G(0, j)$	0	1	2	3	4	5	6	6	6	7	7	7	7	7	7	7
$C_G(1, j)$	0	0	1	2	3	4	5	5	6	7	7	7	7	7	7	7
$C_G(2, j)$	0	0	0	1	2	3	4	5	6	7	7	7	7	7	7	7
$C_G(3, j)$	0	0	0	0	1	2	3	4	5	6	6	6	6	6	6	6
$C_G(4, j)$	0	0	0	0	0	1	2	3	4	5	6	6	6	6	6	6
$C_G(5, j)$	0	0	0	0	0	0	1	2	3	4	5	6	6	6	6	6
$C_G(6, j)$	0	0	0	0	0	0	0	1	2	3	4	5	5	5	6	6
$C_G(7, j)$	0	0	0	0	0	0	0	0	1	2	3	4	5	5	6	6
$C_G(8, j)$	0	0	0	0	0	0	0	0	0	1	2	3	4	5	6	6
$C_G(9, j)$	0	0	0	0	0	0	0	0	0	0	1	2	3	4	5	5
$C_G(10, j)$	0	0	0	0	0	0	0	0	0	0	0	1	2	3	4	5
$C_G(11, j)$	0	0	0	0	0	0	0	0	0	0	0	0	1	2	3	4
$C_G(12, j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	3
$C_G(13, j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2
$C_G(14, j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
$C_G(15, j)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabela 5.1: C_G referente ao GDAG da Figura 5.1.

Propriedades 5.2.1 *Para* $0 \leq i \leq n$,

1. *Existe um único componente finito de* D_G^i *que não aparece em* D_G^{i+1}, *que é* $D_G(i, 0) = i$.
2. *Existe no máximo um componente finito* D_G^{i+1} *que não aparece em* D_G^i.

As Propriedades 5.2.1(1) e 5.2.1(2) indicam que pode-se transformar uma linha D_G em sua linha seguinte através da remoção de um componente (o primeiro) e inserção de um novo componente (finito ou infinito). Diz-se então que duas linhas consecutivas de D_G são ***1-variantes***. Assim se tomar $r + 1$ linhas consecutivas tem-se que elas são *r-variantes*, pois de qualquer uma destas linhas pode-se obter a outra através da remoção e inserção de no máximo r componentes.

Estas propriedades sugerem uma representação em espaço $O(m + n)$ para D_G. Esta representação é composta por D_G^0 (a primeira linha de D_G), que ocupa espaço

	k												
	0	1	2	3	4	5	6	7	8	9	10	11	12
$D_G(0, k)$	0	1	2	3	4	5	6	9	∞	∞	∞	∞	∞
$D_G(1, k)$	1	2	3	4	5	6	8	9	∞	∞	∞	∞	∞
$D_G(2, k)$	2	3	4	5	6	7	8	9	∞	∞	∞	∞	∞
$D_G(3, k)$	3	4	5	6	7	8	9	∞	∞	∞	∞	∞	∞
$D_G(4, k)$	4	5	6	7	8	9	10	∞	∞	∞	∞	∞	∞
$D_G(5, k)$	5	6	7	8	9	10	11	∞	∞	∞	∞	∞	∞
$D_G(6, k)$	6	7	8	9	10	11	14	∞	∞	∞	∞	∞	∞
$D_G(7, k)$	7	8	9	10	11	12	14	∞	∞	∞	∞	∞	∞
$D_G(8, k)$	8	9	10	11	12	13	14	∞	∞	∞	∞	∞	∞
$D_G(9, k)$	9	10	11	12	13	14	∞	∞	∞	∞	∞	∞	∞
$D_G(10, k)$	10	11	12	13	14	15	∞	∞	∞	∞	∞	∞	∞
$D_G(11, k)$	11	12	13	14	15	∞	∞	∞	∞	∞	∞	∞	∞
$D_G(12, k)$	12	13	14	15	∞	∞	∞	∞	∞	∞	∞	∞	∞
$D_G(13, k)$	13	14	15	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$D_G(14, k)$	14	15	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$D_G(15, k)$	15	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Tabela 5.2: D_G referente ao GDAG da Figura 5.1.

$O(m)$, e por um vetor V_G de tamanho $O(n)$, onde $V_G(i)$, $1 \leq i \leq n$, é o valor do componente finito que está presente na linha D_G^i mas não está na linha D_G^{i-1}. Se não houver tal componente finito, $V_G(i) = \infty$. A importância dessas propriedades para o algoritmo está no espaço que as estruturas D_G^0 e V_G ocupam. A Tabela 5.3 mostra os valores de V_G.

Um algoritmo sequencial, baseado nos resultados de [Sch98], é utilizado para resolver o problema ALCS. Dadas como entradas para esse algoritmo duas sequências X e Y, de comprimento m e n respectivamente, tem-se como saída os vetores D_G^0 e V_G relativos às sequências X e Y. Pode-se resolver o problema ALCS sequencialmente em tempo $O(mn)$ e espaço $O(m+n)$ (ou $O(mn)$, se além dos comprimentos também quiser obter as subsequências). Todos os detalhes deste algoritmo são mostrados em [ACS03].

A estrutura composta por D_G^0 e V_G é importante também no algoritmo paralelo, pois a estratégia a ser usada envolve a divisão do GDAG original em faixas e a

	k														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$V_G(k)$	8	7	∞	10	11	14	12	13	∞	15	∞	∞	∞	∞	∞

Tabela 5.3: V_G referente ao GDAG da Figura 5.1.

resolução do ALCS em cada faixa. Estas faixas são então unidas, o que requer o envio da resposta dos ALCS parciais entre os processadores. Na seção 5.2.1 é apresentado o algoritmo paralelo para o problema ALCS.

5.2.1 Algoritmo CGM para o Problema ALCS

O algoritmo CGM para o problema ALCS desenvolvido por Alves et al. [ACS03] com p processadores, requer tempo $O(\frac{mn}{p})$ e $O(\log p)$ rodadas de comunicação, sendo que em cada rodada cada processador envia/recebe $O(mp+n)$ dados. Para este algoritmo foi desenvolvida uma estrutura de dados eficiente para representar a solução do problema que ficará distribuída entre os processadores. Esta estrutura pode ser criada em tempo e espaço $O(n\sqrt{m/p})$ e consultada (somente leitura) em tempo constante.

Considere $C_G(i,j)$ o comprimento do maior caminho entre $T_G(i)$ e $F_G(j)$ do GDAG G, $0 \leq i \leq j \leq n$. Tem-se que $C_G(i,j) - C_G(i,j-1)$ é sempre 0 ou 1, basta então registrar os valores de j para os quais tal diferença é 1.

O algoritmo CGM começa pela divisão horizontal do GDAG em p GDAGs menores que são chamados de **faixas**, sendo cada uma delas entregue a um processador. Em cada faixa, o algoritmo sequencial para o ALCS é usado e a estrutura composta por D_G^0 e V_G é gerada, em tempo $O(\frac{mn}{p})$. Em seguida, as faixas são unidas duas a duas em $\log p$ etapas e envia a resposta dos ALCS parciais entre os processadores. Observe a Figura 5.2 que mostra a divisão horizontal do GDAG G em 8 processadores e depois a união das faixas. No restante da seção será brevemente descrito como é feita a união entre duas faixas.

Sejam A e B duas faixas que são unidas em uma faixa U e o número de processadores envolvidos é r. A e B são GDAGs de altura d e largura $e = n$.

Os dados de D_A e D_B são recebidos pelos r processadores em suas estruturas de dados. Cada processador constrói localmente as estruturas de acesso em tempo constante, o que requer tempo e espaço local $O(e\sqrt{d})$.

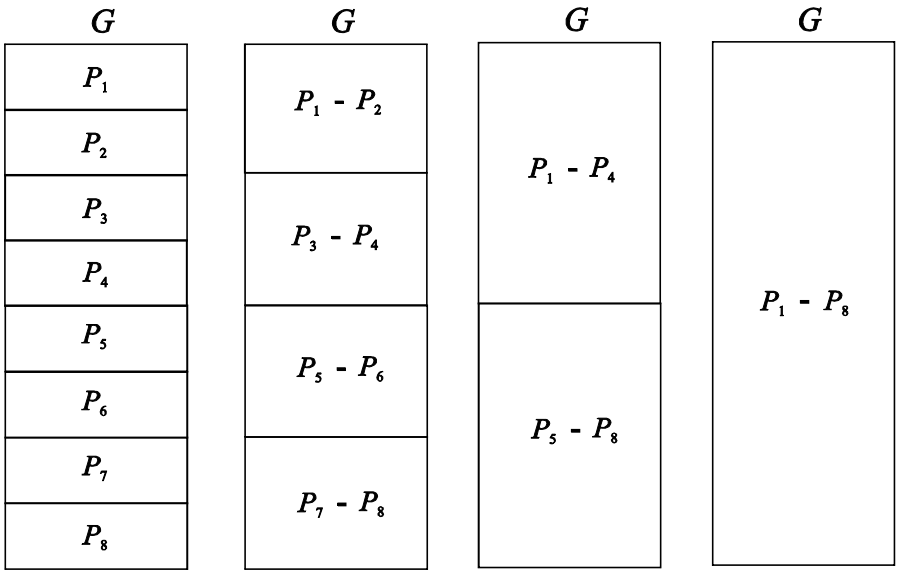


Figura 5.2: Distribuição do GDAG entre 8 processadores e união das faixas.

D_A possui linhas contíguas muitos semelhantes (1-variantes), com isso para determinar as linhas contíguas de D_U uma série de matrizes *totalmente monotônicas* são envolvidas, também muitos semelhantes, com partes que são comuns a todas elas. Em [ACS03] encontram-se descritas as características das matrizes totalmente monotônicas, que dizem respeito à localização dos elementos mínimos em cada coluna. Aproveitando a semelhança entre as matrizes, cada processador trabalha com um bloco de aproximadamente $\sqrt{d}$ linhas contíguas de D_U de cada vez. Assim as partes comuns a todas as matrizes totalmente monotônicas envolvidas são determinadas e contraídas (cada uma é substituída por uma linha que contém os mínimos de suas colunas).

Para determinar $D_U^i(k)$, sendo $D_U^i(k) = D_U(i,k)$, note que todos os caminhos a partir de $T_U(i) = T_A(i)$ até $F_U(j) = F_B(j)$ têm que cruzar a fronteira $F_A = T_B$ em algum vértice e o peso total do caminho é a soma dos pesos em A e em B, ou seja, é preciso considerar os caminhos que cruzam A com peso l e B com peso $k-l$, $0 \leq l \leq d$. Em outras palavras tem-se que:

$$D_U(i,k) = \min_{0 \leq l \leq d} \{D_B(D_A(i,l), k-l)\}.$$

Considere a determinação de uma linha D_U^i, $0 \leq i \leq e$. Para determinar D_U^i, pode-se construir uma matriz cujas linhas são cópias deslocadas de linhas de D_B,

usando D_A^i para escolher que linhas de D_B tomar. A matriz construída tem a propriedade totalmente monotônicas que diz respeito a encontrar os mínimos de todas as suas colunas.

Para construir essa matriz é preciso determinar o deslocamento das linhas, para isso veja as seguintes definições: $Desl[l, W, c]$ é o vetor W deslocado para a direita l posições e completado com ∞, onde W é um vetor de comprimento $s + 1$ (índices de 0 a s), para qualquer l variando de $0 \leq l \leq c - s$. Esse vetor W armazena os índices das linhas de serão copiadas para uma matriz. $Diag[W, M, l_0]$ é a matriz de dimensões $(d' + 1) \times (2d + 1)$, que tem suas linhas sendo cópias feitas a partir de uma matriz M, onde M é uma matriz $(e + 1) \times (d + 1)$ (índices iniciais 0). A seleção de quais linhas são copiadas é feita pelo vetor W de números inteiros em ordem crescente tal que os $d' + 1$ primeiros componentes são finitos e $W(d') \leq e$. Cada linha que é copiada é deslocada uma coluna para a direita em relação à linha anterior. O deslocamento da primeira linha copiada é dado por l_0.

Tem-se $MD[i] = MD[U, i]$. Seja U um GDAG para o problema ALCS, formado pela união dos GDAGs A (superior) e B (inferior). Então $MD[i] = Diag[D_A^i, D_B, 0]$ mostrado na Tabela 5.4. Para o exemplo mostrado nesta tabela foi usado o GDAG U, formado por duas cópias do GDAG da Figura 5.1. O D_A e o D_B são iguais dados pela Tabela 5.2. O $MD[1]$ tem suas linhas definidas por $D_A^1 = (1, 2, 3, 4, 5, 6, 8, 9)$.

	k															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15-24
MD[1](0, *k*)	1	2	3	4	5	6	8	9	∞	∞	∞	∞	∞	∞	∞	∞
MD[1](1, *k*)	∞	2	3	4	5	6	7	8	9	∞	∞	∞	∞	∞	∞	∞
MD[1](2, *k*)	∞	∞	3	4	5	6	7	8	9	∞	∞	∞	∞	∞	∞	∞
MD[1](3, *k*)	∞	∞	∞	4	5	6	7	8	9	10	∞	∞	∞	∞	∞	∞
MD[1](4, *k*)	∞	∞	∞	∞	5	6	7	8	9	10	11	∞	∞	∞	∞	∞
MD[1](5, *k*)	∞	∞	∞	∞	∞	6	7	8	9	10	11	14	∞	∞	∞	∞
MD[1](6, *k*)	∞	∞	∞	∞	∞	∞	8	9	10	11	12	13	14	∞	∞	∞
MD[1](7, *k*)	∞	∞	∞	∞	∞	∞	∞	9	10	11	12	13	14	∞	∞	∞

Tabela 5.4: $MD[1]$ do GDAG formado pela união de dois GDAGs iguais da Figura 5.1.

A determinação de cada linha de D_U^i envolve uma matriz totalmente monotônicas que pode ser compactada em $O(\sqrt{d})$ linhas, devido às partes comuns das matrizes que foram contraídas. Essa compactação pode ser vista em [ACS03]. Resolve-se o problema dos mínimos das colunas em $MD[i]$ encontrando assim o D_U^i, tem-se:

$$D_U^i(k) = Cmin[MD[i]](k)$$

onde:

Cmin[*M*] é o vetor de mínimos das colunas da matriz M. Nesse caso, D_U^1 é $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, \infty)$.

O vetor V_U pode ser enviado a todos os processadores que irão usar D_U na próxima etapa de união. Um dos processadores determina e envia D_U^0 para todos os outros processadores completando a distribuição da estrutura de D_U.

Depois de $\log p$ etapas de união, a estrutura D_G está completa nas memórias de todos os processadores, sendo estendida para estrutura de tamanho $O(e\sqrt{d})$ permitindo consultas rápidas. Com as estruturas de dados mantidas ao longo de todas as etapas do processo, pode-se ao final obter qualquer caminho no GDAG G. Permitindo a obtenção da maior subsequência comum entre a X e Y, resultando em um algoritmo CGM para o problema LCS, com complexidade $O\left(\frac{mn}{p}\right)$ e apenas $O(\log p)$ rodadas de comunicação.

Capítulo 6

Algoritmo Paralelo PRAM para o problema LCS

Neste capítulo, é apresentado outra solução para o problema da Maior Subsequência Comum usando o modelo PRAM [Lu90]. A ideia do algoritmo é encontrar a maior subsequência comum, baseado na técnica da divisão e conquista, a qual permite concorrência e portanto eficiência. O algoritmo é executado em r processadores, onde r é o número total de casamentos entre duas sequências. Realiza um pré-processamento para encontrar os r casamentos, que é executado em $m+n$ processadores. Necessita de $O(\log \rho \log^2 n)$ no tempo de execução, onde ρ é a maior subsequência comum e n é o tamanho da maior sequência entre as duas sequências.

Dadas as sequências $X = x_1 x_2 \cdots x_i \cdots x_m$ e $Y = y_1 y_2 \cdots y_i \cdots y_n$ como entrada do problema LCS e o tamanho da sequência $|X| = m$ e $|Y| = n$, sendo $m \leq n$. Seja dada $c(i, j)$ o tamanho da LCS de $x_1 x_2 \ldots x_i \ldots x_m$ e $y_1 y_2 \ldots y_i \ldots y_n$, onde $c(i, j)$ pode ser determinada conforme já descrito na seção 3.2.1.

Uma matriz c, de dimensão $m \times n$, é construída para calcular $c(i, j)$. Os $c(i, j)$, com $i \leq 0 \leq m$ e $i \leq j \leq n$, são calculados linha por linha e o tamanho da LCS é dado por $c(m, n)$ quando a computação é concluída. A seguir são descritos os detalhes do algoritmo do pré-processamento.

6.1 Pré-processamento

O pré-processamento é a etapa anterior à execução do algoritmo LCS. Seu objetivo é identificar os r pontos tal que $x_i = y_j$, ou seja, os símbolos que se casam. Dadas duas sequências X e Y, onde o tamanho de X é m (linhas) e o tamanho de Y é n (colunas). Suponha que $m \leq n$ sem perda de generalidade. O par (i, j) indica o casamento de dois símbolos na posição i na sequência X e posição j na sequência Y. Para encontrar os (i, j) pares, são necessários m processadores para atender os símbolos na sequência X e n para a sequência Y. Distribui-se os símbolos em $m + n$ processadores, um símbolo por processador. Supõe-se que esses símbolos têm uma ordem lexicográfica. Cada processador que contém um símbolo de X gera um registro com dois campos: $[i, ordem]$ e os que contêm símbolos em Y geram os registros $[j, ordem]$. Os i e j indicam a posição do símbolo na sequência X e Y, respectivamente, e *ordem* indica a posição na ordem lexicográfica do símbolo contido no processador. Abaixo encontram-se as etapas (operações executadas) do pré-processamento:

1. Ordenar pelo campo *ordem* os m símbolos de X usando m processadores. Os empates são quebrados pelo índice i. O índice i_1 é a posição inicial de cada símbolo em sua sequência original, após a ordenação dos símbolos tem-se a sequência ordenada e cada símbolo tem um novo índice i_2, mas o índice i_1 não é descartado. Veja a Figura 6.1.

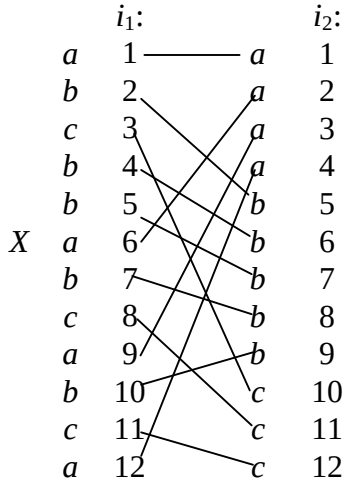


Figura 6.1: Ordenação dos m símbolos de X na segunda coluna, representada pelo índice i_2.

2. Selecionar os símbolos distintos cada um com menor índice i_2 na sequência ordenada para formar uma sequência concentrada. A sequência concentrada também recebe o seu índice, i_3. Assume-se que há no total t símbolos distintos. Observe a Figura 6.2.

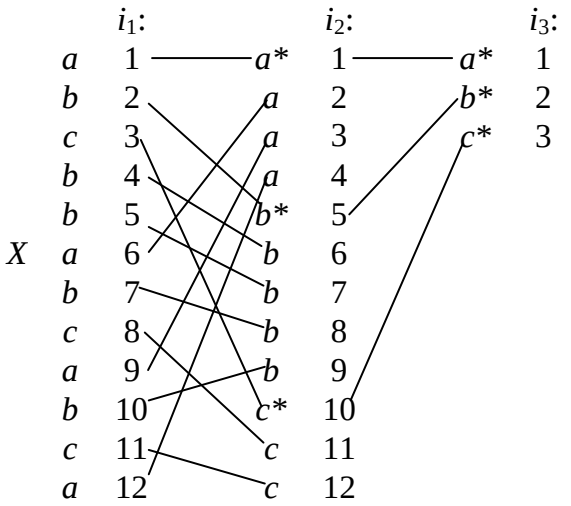


Figura 6.2: Seleção de símbolos distintos representado na terceira coluna pelo índice i_3.

3. Cada processador que contém um símbolo da sequência Y executa uma busca binária nos t registros da sequência concentrada, para encontrar a ordem a qual esse símbolo da sequência Y casa com a ordem mantida em seu registro, observe a Figura 6.3.

								Y							
	c	b	a	c	b	a	a	b	a	b	a	c	b	a	a
j:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i_3:	3	2	1	3	2	1	1	2	1	2	1	3	2	1	1

Figura 6.3: Busca binária nos t registros da sequência concentrada.

4. Calcular a diferença sobre o índice i_2 da sequência ordenada mostrado na Figura 6.1. O cálculo é feito somente entre os símbolos distintos que são adjacentes. Neste cálculo usa-se os valores do índice i_2 entre os símbolos (a, b, c) que aparecem pela primeira vez na sequência ordenada. Considerando que

a sequência está ordenada tem-se condições de saber a quantidade de cada símbolo distinto da sequência X. Veja a seguir os detalhes desta operação:

(a) No passo 1 o símbolo b aparece na posição 5 do índice i_2 e o símbolo a aparece na posição 1 do índice i_2, calculando a diferença tem-se $5-1=4$, isso significa que existem 4 símbolos a.

(b) O símbolo c aparece na posição 10 do índice i_2 e o símbolo b aparece na posição 5 do índice i_2, calculando a diferença tem-se $10-5=5$, isso significa que tem 5 símbolos b.

(c) Para encontrar a quantidade do último símbolo o cálculo efetuado difere um pouco dos anteriores, antes do cálculo da diferença soma-se 1 ao valor da última posição do índice i_2, neste caso a última posição é 12, calculando a soma tem-se $12+1=13$, a partir deste valor, 13, calcula-se a diferença com a primeira posição do índice i_2 do símbolo c, $13-10=3$, isso significa que tem 3 símbolos c. Os asteriscos apresentados na Figura 6.4 representam a quantidade de cada símbolo da sequência X.

								Y							
	c	b	a	c	b	a	a	b	a	b	a	c	b	a	a
j:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
		*	*		*	*	*	*	*	*	*		*	*	*
		*			*			*		*			*		

Figura 6.4: Número de casamentos.

5. Executar uma soma prefixa da computação paralela de modo que cada símbolo da sequência Y em sua posição j obtém o número total, a saber c_j, de casamentos encontrados para todos os símbolos antes dele. Preparar r processadores para executar o algoritmo LCS se r casamentos são encontrados no total. Cada processador mantendo um símbolo em Y notifica o processador c_j entre r processadores a posição j em Y, o símbolo é atribuído e a posição do símbolo na sequência ordenada de Y. Veja a tabela 6.1.

6. Usando o traçado inverso nos índices $i_3 \rightarrow i_2 \rightarrow i_1$, cada um dos r processadores pode obter a informação sobre a posição i em X quando o símbolo casa:

								Y							
j:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	8	12	15	20	24	28	33	37	42	46	49	54	58	62

Tabela 6.1: Soma prefixa.

(a) Primeiramente, atribui a posição j do símbolo em Y para cada processador, por exemplo, observe a Figura 6.4, o símbolo c está na posição $j = 1$ e esse símbolo casa três vezes, então os processadores $1, 2$ e 3 recebem cada um a posição 1. O símbolo b está na posição $j = 2$ e casa cinco vezes, consequentemente os processadores $4, 5, 6, 7$ e 8 recebem cada um a posição 2 e assim sucessivamente até finalizar o último símbolo em Y.

(b) Após os processadores obterem essa informação, em um segundo momento é atribuído aos processadores qual a posição i_3 deste símbolo. Analisando o mesmo exemplo, na Figura 6.5 têm-se o seguinte: o símbolo c ocupando os processadores $1, 2$ e 3 que contêm a informação 1 em cada um dos respectivos processadores e nesta etapa recebem a posição 3, pois é a posição que c ocupa na sequência concentrada i_3. Da mesma maneira, o símbolo b ocupa os processadores $4, 5, 6, 7$ e 8 que contém a informação 2 cada um e agora recebem a posição 2, posição que b ocupa em i_3 e assim por diante.

(c) Conforme a descrição acima o mesmo ocorre para i_2 e i_1, onde a partir da posição i_3 procura-se pela posição i_2 e depois pela posição i_1 de cada um dos símbolos. Obtendo a informação i_1 tem-se a posição i que o símbolo de X casa com o símbolo de Y. Observe o resultado do passo 6 na Figura 6.5 e confira este resultado com a Tabela 6.2.

PE:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
j:	**1**	**1**	**1**	**2**	**2**	**2**	**2**	**2**	**3**	**3**	**3**	**3**	**4**	**4**	**4**	**5**	**5**	**5**	**5**	**5**	**6**	**6**	**6**	**6**	**7**	**7**	**7**	**7**	**8**	**8**	**8**
i_3:	3	3	3	2	2	2	2	2	1	1	1	1	3	3	3	2	2	2	2	2	1	1	1	1	1	1	1	1	2	2	2
i_2:	10	11	12	5	6	7	8	9	1	2	3	4	10	11	12	5	6	7	8	9	1	2	3	4	1	2	3	4	5	6	7
i_1:	**3**	**9**	**13**	**2**	**5**	**6**	**8**	**12**	**1**	**7**	**11**	**14**	**3**	**9**	**13**	**2**	**5**	**6**	**8**	**12**	**1**	**7**	**11**	**14**	**1**	**7**	**11**	**14**	**2**	**5**	**6**

PE:	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62
j:	**8**	**8**	**9**	**9**	**9**	**9**	**10**	**10**	**10**	**10**	**10**	**11**	**11**	**11**	**11**	**12**	**12**	**12**	**13**	**13**	**13**	**13**	**13**	**14**	**14**	**14**	**14**	**15**	**15**	**15**	**15**
i_3:	2	2	1	1	1	1	2	2	2	2	2	1	1	1	1	3	3	3	2	2	2	2	2	1	1	1	1	1	1	1	1
i_2:	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	10	11	12	5	6	7	8	9	1	2	3	4	1	2	3	4
i_1:	**8**	**12**	**1**	**7**	**11**	**14**	**2**	**5**	**6**	**8**	**12**	**1**	**7**	**11**	**14**	**3**	**9**	**13**	**2**	**5**	**6**	**8**	**12**	**1**	**7**	**11**	**14**	**1**	**7**	**11**	**14**

Figura 6.5: Traçado inverso.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			c	b	a	c	b	a	a	b	a	b	a	c	b	a	a
0																	
1	a				*			*	*		*		*			*	*
2	b			*			*			*		*			*		
3	c		*			*								*			
4	b			*			*			*		*			*		
5	b			*			*			*		*			*		
6	a				*			*	*		*		*			*	*
7	b			*			*			*		*			*		
8	c		*			*								*			
9	a				*			*	*		*		*			*	*
10	b			*			*			*		*			*		
11	c		*			*								*			
12	a				*			*	*		*		*			*	*

Tabela 6.2: Tabela que apresenta o casamento dos símbolos da sequência X com os símbolos da sequência Y.

Com o término da etapa do traçado inverso o pré-processamento está finalizado, têm-se todos os casamentos entre a sequência X e a sequência Y. Na próxima seção é descrita a etapa do processamento.

6.2 Processamento

Na etapa do processamento, cada casamento $c(i, j)$ do pré-processamento apresentado anteriormente é referenciado como **ponto** p e o valor $c(i, j)$ correspondente é chamado de **classe do ponto**, denotado como $classe(p)$. O ponto mais à esquerda de uma linha na classe é denotado como o **ponto de quebra** se ele é o ponto superior de sua coluna na classe.

As linhas horizontais e verticais que conectam os pontos da mesma classe formam o **contorno da classe**. Observe na Figura 6.6. O ponto $q(i_q, j_q)$ é **dominado** pelo ponto $p = (i_p, j_p)$ se $i_p < i_q$ e $j_p < j_q$. Este conceito de dominação é usado em vários momentos no algoritmo.

Veja a seguir alguns passos que descrevem a etapa do processamento:

1. Os r pontos encontrados durante a etapa do pré-processamento usam r proces-

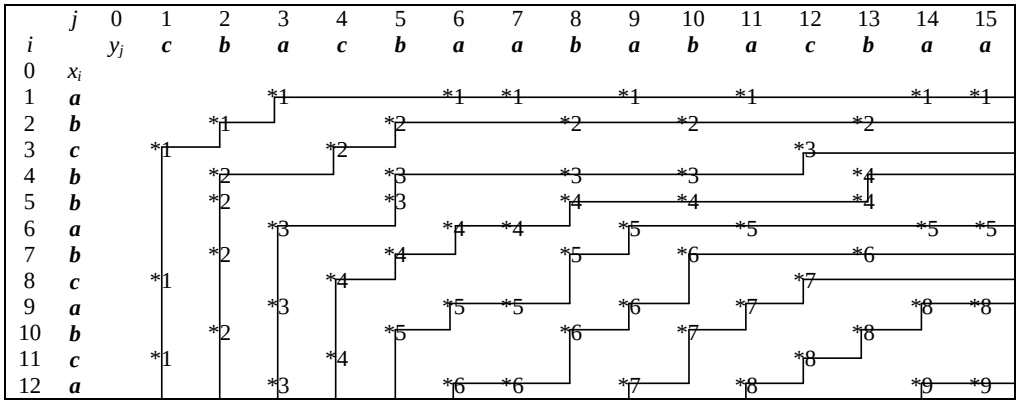


Figura 6.6: Contorno da classe.

sadores no modelo PRAM. Os r pontos são divididos por uma linha horizontal em duas metades, tal que a metade inferior tem valores de i maiores que os pontos na metade superior.

2. Recursivamente resolve o problema para cada metade de forma concorrente.

3. Os resultados em suas metades precisam juntar-se. É necessário fazer uma intercalação dos dois resultados. Antes da intercalação, cada processador conhece a classe de seus pontos em sua metade. Depois da intercalação, as classes dos pontos na metade superior ficam inalterados e aqueles na metade inferior terão que recomputar suas classes. A etapa da intercalação inclui duas fases em que a classe de um ponto é atualizada:

   (a) Atualização com Junção: Seja $junção(k)$, o ponto mais a esquerda de cada classe k. A $junção(k)$ é projetada na linha horizontal, criando as junções. A junção 0 é inserida à esquerda das demais junções. A atualização com junção é feita dos pontos da metade superior com os pontos da metade inferior. É necessário recomputar as classes dos pontos da metade inferior, de acordo com as junções projetadas na linha divisora.

   (b) Atualização entre Árvores: Em cada iteração i, 2^i árvores são intercaladas. Denote as 2^{i-1} árvores da esquerda por **árvores esquerda** e por **árvores direitas** as 2^{i-1} árvores direitas. Cada ponto $p(i_p, j_p)$ deve encontrar na árvore direita a ele o ponto $t(i_t, j_t)$ tal que o ponto t domina o ponto p e a $classe(t)$ é a máxima.

A seguir será apresentado o algoritmo do processamento, que detalha os passos anteriormente descritos.

6.2.1 Algoritmo do Processamento

O algoritmo do processamento inicia com a distribuição dos r pontos em r processadores de modo que cada processador contenha um ponto. Baseado na técnica de divisão e conquista, a classe de cada ponto na metade superior ou inferior da linha divisora é identificada antes da intercalação e recursivamente resolve o problema para cada metade. Observe na Figura 6.7 a divisão dos pontos em duas metades.

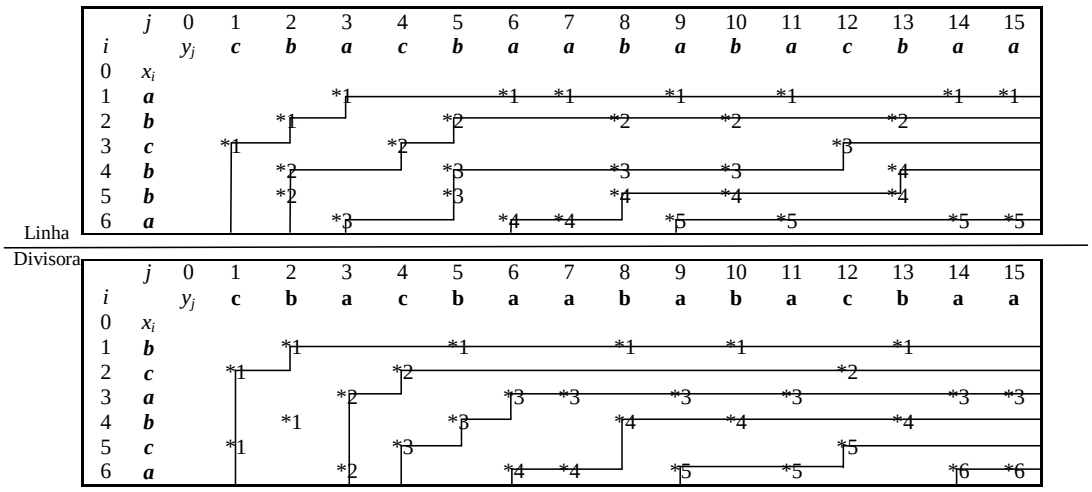


Figura 6.7: Divisão dos r pontos em duas metades.

Cada processador que mantém um ponto mais à esquerda de uma classe na metade superior mantém uma junção, que é um ponto com o mesmo valor de j mas com o valor de i igual da linha divisora. A classe de um ponto é considerada como o índice da junção. Na Figura 6.8 pode-se identificar os índices das junções. Observe a junção 0 criada à esquerda das demais junções.

Na Atualização com Junção cada ponto da metade inferior encontra seu pai na classe acima. Define-se um ponto $f(i_f, j_f)$ como sendo o ponto pai de $s(i_s, j_s)$, se o ponto f domina o ponto s e a $classe(f) = classe(s) - 1$. Inicialmente, escolhe-se para cada ponto seu pai na classe acima dele e com j_f o mais próximo dos j do que todos os outros pontos a $classe(f)$. Um ponto na classe 1 busca seu pai entre as junções. Árvores são formadas onde as junções são as raízes. Veja a Figura 6.9.

Os pontos calculam o $índice(classe)$ da raiz e sua profundidade na árvore e decidem sua nova classe, essa subrotina é chamada de Propagação 1.

Na Propagação 1 a raiz transfere seu índice a seu filho, o filho dos filhos e assim

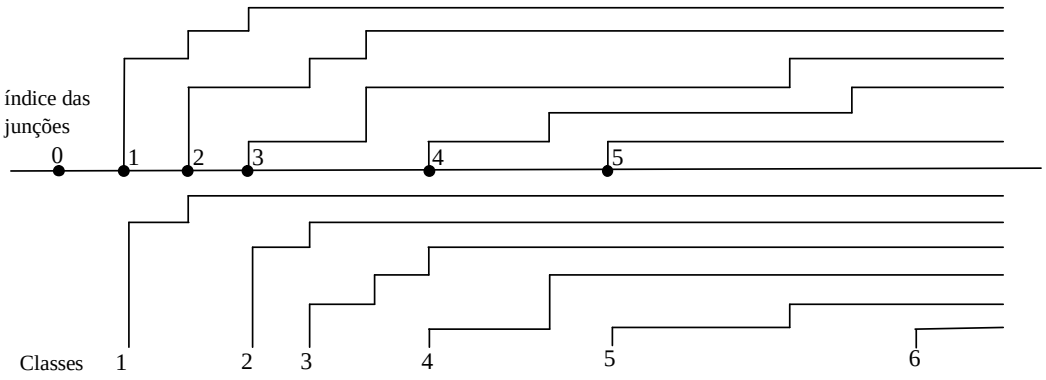


Figura 6.8: Projeção dos pontos mais à esquerda na linha divisora formando os índices das junções.

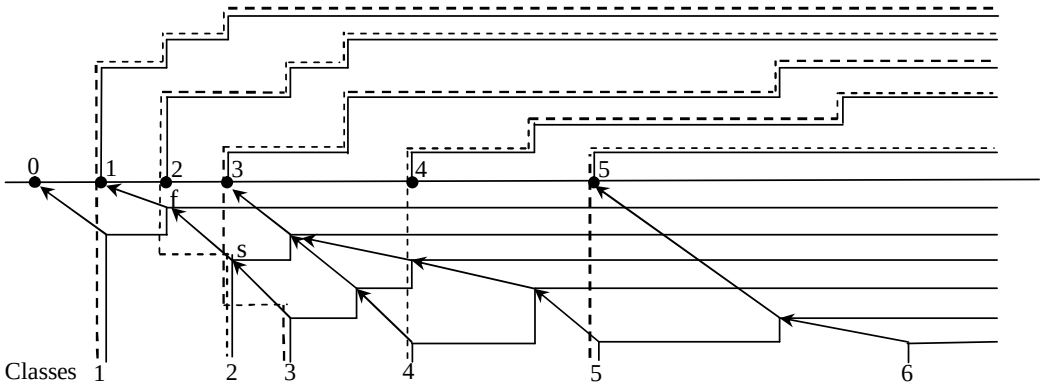


Figura 6.9: Atualização com Junção.

sucessivamente, ou seja, é realizado o *pointer jumping* [CMS01]. Exemplificando, se 2 é o índice da raiz de uma árvore de profundidade 8, ela transfere para seu filho o seu índice, que é sua classe, o filho por sua vez recebe esse dado e acrescenta 1, isto é, a classe do filho é atualizada para 3 ($2+1=3$). Então a raiz e seu filho transferem para os filhos de seus filhos suas classes respectivamente, 2 e 3. É acrescentado 2 ao dado recebido para o neto da raiz e para o neto do filho, com isso a classe dos netos é atualizada. O neto da raiz passa a ter a classe 4 e o neto do filho passa a ter a classe 5. A Figura 6.10 demonstra como a propagação continua para todos os níveis da árvore, até que todos sejam atualizados.

Depois de completada a fase da Atualização com Junção é necessário fazer a Atualização entre Árvores. Os pontos são ordenados em cada árvore pelo valor de j. Após a ordenação é executado a sub-rotina Premax. Nesta etapa do algoritmo, é

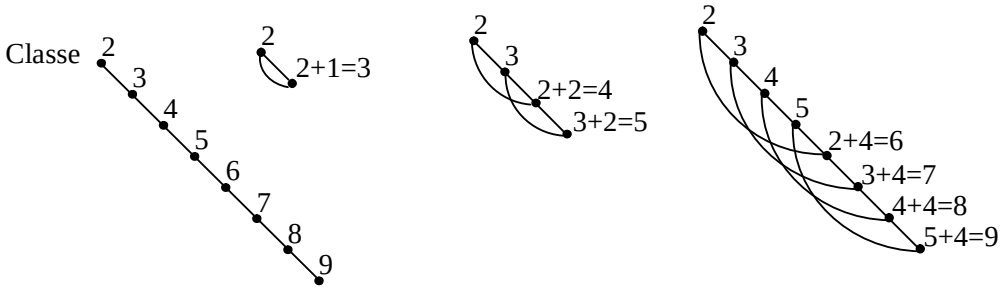


Figura 6.10: Propagação 1.

feita uma comparação recursiva entre todos os pontos para identificar a maior classe entre os pontos.

Sejam $C_0, C_1, \cdots, C_n$ os itens dos n dados de uma lista. Encontra-se $M(k) = \max_{i=0}^{k}(C_i)$, para $k = 0, 1, \cdots, n-1$. Tem-se $M(0) = C_0$, e $M(k) = \max\{M(k-1), C_k\}$, para $k = 0, 1, \cdots, n-1$. É necessário distribuir os dados nos n processadores. No exemplo da Figura 6.10 tem-se $n = 8$. A comparação recursiva é executada em vários passos. No primeiro passo, cada C_i mantido no processador i é comparado com C_{i+1} e o resultado desta comparação é armazenado no processador $i+1$, isto acontece para $i = 0, 1, \cdots, 6$. No segundo passo, o resultado armazenado no processador i é comparado com o resultado armazenado do processador $i+2$, para $i = 0, 1, \cdots, 5$. No último passo, o resultado armazenado no processador i é comparado com o valor que está no processador $i+4$, para $i = 0, 1, \cdots, 3$. Consequentemente, o processador k terá $M(k)$ como o resultado final, para $k = 1, 2, \cdots, n-1$. Serão necessários $O(\log n)$ passos para completar os n itens de dados da lista.

Com o resultado do Premax, cada ponto, $p(i_p, j_p)$ deve encontrar na árvore direita a ele, o ponto $t(i_t, j_t)$ tal que o ponto t domina o ponto p e a $classe(t)$ é máxima. Quando o algoritmo encontra este ponto ele chama a subrotina da Propagação 2.

A Propagação 2 difere da Propagação 1 no que diz respeito a qual raiz um determinado ponto pertence para que possa atualizar sua classe. Várias árvores podem ser formadas após a intercalação e no momento da propagação um determinado ponto tem dois possíveis pais, então é necessário com a Propagação 2 definir o pai desse ponto. Cada ponto compara os dados recebidos com o dado que possui e determina se aceita os dados ou não. O ponto troca de pai, se a classe deste pai for maior que a classe do pai que ele tem no momento. Neste caso é feita a Atualização entre Árvores, onde todos os pontos sem distinção passam por esse processo de atualização.

No algoritmo, não serão os ancestrais que vão transferir seu índice para seus descendentes, mas os descendentes farão o pedido dos dados para seus ancestrais. Cada ponto tem o endereço do seu pai. Num primeiro momento, o ponto pede o endereço do pai do pai dele e num segundo momento o ponto pede o endereço do avô do avô. Para ambas propagações 1 e 2, $\log \rho$ iterações são necessárias para completar a propagação ao longo de uma árvore de profundidade ρ.

6.3 O algoritmo e a complexidade de tempo

O algoritmo da intercalação é descrito a seguir.

Algoritmo 6.3.1 *LCS*

1. Ordenar os pontos em cada classe contorno pelo j independentemente e concorrentemente.
/*Atualização com Junção*/
2. Cada ponto na parte debaixo encontra seu pai na classe acima dele; para aqueles pontos na classe 1, cada um encontra seu pai por entre as junções. Árvores são formadas. Atribui *flag(junção)* para 1, e *classe(junção)* para ser o índice da junção.
3. Executar *Propagação*1*(classe)* em cada árvore.
/*Atualização entre Árvores*/
4. **faça** $\log(\rho + 1)$ vezes (com $\rho = \max(classe(junção))$
5. **início**
6. Ordenar todos os pontos na árvore pelo valor j.
7. Executar $Premax(classe, M_c, Id)$ em cada árvore.
8. Cada ponto $p(i_p, j_p)$ encontra nas árvores à direita o ponto $t(i_t, j_t)$ tal que $j_t < j_p$ e mais próximo de j_p. Obter $M_c(t)$ e $Id(t)$ do ponto t, e então registra como $temp_1(p) = M_c(t)$, $temp_2(p) = Id(t)$.
9. $dif(p) = temp_1(p) - classe(p)$.
10. **se** $dif(p) > dif(pai(p))$ **então** atribui $flag(p) = 1$ e Executa a *Propagação 2 (dif)*.
11. $classe(p) = classe(p) + dif(p) + 1$.
12. **se** $dif(p) > dif(pai(p))$ **então** $pai(p) = temp_2(P)$;
13. **fim**
14. **fim** /*LCS*/

A subrotina *Propagação*1 é apresentada a seguir:

Algoritmo 6.3.2 *Propagação 1 (classe)*

1. Cada ponto p executa $end = pai(p)$.
2. **Para** $i = 1$ **até** $\log \rho$ **faça** (com ρ sendo a profundidade máxima da árvore).
3. **início**
4. Cada ponto faz um pedido do endereço do pai.
5. **se** $flag(end) = 1$ **então**
6. **início**
7. $temp(p) = classe(end)$;
8. $classe(p) = temp(p) + 2^{i-1}$;
9. $flag(p) = 1$.
10. **fim**
11. $end = end(end)$
12. **fim**
13. **fim** /*Propagação 1*/

O Algoritmo *Propagação 2 (valor)* é similar ao *Propagação 1 (classe)*, mas com o passo 3 e 4 modificado como segue:

3'. $temp(p) = valor(end)$;
4'. **se** $temp(p) > valor(p)$ **então** $valor(p) = temp(p)$;

Seguindo está o algoritmo do *Premax*.

Algoritmo 6.3.3 *Premax(classe, M_c, Id)*

1. Cada processador i executa $M_c(i) = classe(i)$.
2. **Para** $k = 0$ **até** $\log k - 1$, **faça**
3. **início**
4. Todos os i processadores caminha para $M_c(i)$ e i para os processadores $(i + 2^k)$. Todos os processadores registram o dado recebido em $temp_1(i)$ e $temp_2(i)$ respectivamente.
5. Os i processadores com $i > 2^k - 1$ executam
6. **se** $temp_1(i) > M_c(i)$, **então**
7. **início**
8. $M_c(i) = temp_1(i)$
9. $Id(i) = temp_2(i)$
10. **fim**

11. **fim**
12. **fim** */*Premax**/*

Após a apresentação do algoritmo, faz-se necessário avaliar o tempo de execução para que se possa analisar o desempenho e eficiência do mesmo.

Executando em r processadores, com r igual ao número total de casamentos entre dois símbolos, a complexidade de tempo do algoritmo LCS foi analisada da seguinte forma. O número máximo de pontos na classe contorno é de $m + n$, com $n \geq m$. Consequentemente a ordenação na etapa 1 leva $O(\log n)$ conforme [Lu90]. A etapa 2 envolve uma busca binária na classe contorno, então para o pior caso tem-se um tempo $O(\log n)$. A Propagação incluir $O(\log \rho_1)$ iteração na etapa 3 no pior caso na etapa 8, se ρ_1 é o número total de classes na parte debaixo para ser fundido. A etapa 4 para a etapa 10 é repetido $\log(\rho_2 + 1)$ vezes com ρ_2 igual ao número total de classes na parte de cima para ser fundido. Na etapa 4, ordenar os k pontos nas árvore requer tempo de $O(\log k)$. O Premax envolvido na etapa 5 necessita de $\log k$ passos para encontrar o dado máximo desejado para k dados. Um ponto é buscado na etapa 6 é pesquisado um menor ponto e mais próximo de j na árvore à direita, necessita de um tempo de $O(\log k)$ dado k pontos nas árvores. Para cada uma destas etapas, $k = n$ isso no pior caso, assim, o tempo total necessário é $O(\log \rho \log^2 n)$ com ρ sendo a maior subsequência comum.

Capítulo 7

Algoritmo Paralelo CGM para o problema LCS

Este capítulo apresenta uma solução para o problema da Maior Subsequência Comum usando o modelo CGM. O algoritmo descrito aqui é uma adaptação do algoritmo usado no modelo PRAM apresentado no capítulo 6.

Retomam-se alguns conceitos usados no algoritmo do capítulo anterior. O casamento, $x_i = y_j$, indica que dois símbolos, um na posição i na sequência X e outro na posição j na sequência Y, são iguais. Da mesma forma que no modelo PRAM, denota-se cada casamento, $c(i, j)$, como sendo um ponto p e o valor do correspondente $c(i, j)$ é chamado de classe do ponto, denotado como $classe(p)$.

O conceito de contorno da classe é também mantido neste algoritmo. O contorno é formado pelas linhas horizontais e verticais que conectam os pontos da mesma classe. Lembrando que os pontos que conectam essas linhas são chamados de pontos de quebra. O ponto de quebra é aquele ponto mais à esquerda de uma linha e superior em sua coluna, na classe.

Cada ponto precisa guardar informações como: sua posição (i, j), a posição (i', j') do pai do ponto, sua classe e um *flag*, esta última informação é utilizada nas atualizações que ocorrem no decorrer do algoritmo. Para armazenar essas informações é necessário uma estrutura de árvore além de uma estrutura de matriz. A seguir são descritos os detalhes do algoritmo CGM.

7.1 Processamento no CGM

No algoritmo do modelo PRAM, inicialmente é descrita a etapa do pré-processamento que trata de encontrar os casamentos entre as sequências. No modelo CGM, esta etapa pode ser facilmente realizada. Dadas duas sequências de entrada X e Y pode-se encontrar todos os casamentos entre as sequências X e Y da seguinte forma: o algoritmo envia para todos os processadores a sequência Y inteira e distribui uniformente a sequência X entre os processadores, ou seja, cada processador p_k, $k = 0, \cdots, p-1$, recebe os símbolos $X[k\frac{m}{p}..(k+1)(\frac{m}{p-1})]$. Localmente, aplica-se o algoritmo Comprimento-LCS. Assim somente os símbolos que se casam são os dados necessários para constituir uma LCS. A identificação de cada casamento define um ponto. Todos os casamentos, $c(i,j)'s$, formam o conjunto de r pontos, onde para cada ponto p tem-se a $classe(p)$.

A Figura 7.1 mostra a distribuição da sequência $X = \langle c, c, a, a, c, b, c, b, a, b, a, c, c, a, b, b\rangle$ e $Y = \langle c, a, a, c, b, a, b, a, b, b, c, a, a, b, c, c\rangle$ para 4 processadores. Os casamentos são representados pelo asterisco (*).

Considere a matriz c resultante da aplicação da Comprimento-LCS. O algoritmo usa a técnica de divisão e conquista. Dividem-se os r pontos por uma linha horizontal em duas metades, tal que a metade inferior tem valores de i maiores que os pontos na metade superior. No modelo CGM, os pontos estão distribuídos aleatoriamente entre os processadores. Para que aconteça a divisão de forma balanceada, precisa-se saber a quantidade de pontos. Para encontrar essa quantidade é necessário que seja feita uma soma em cada linha de cada processador armazenando esses dados em um vetor de tamanho $\frac{m}{p}$. Realize em paralelo uma soma prefixa em todos esses vetores. Através dessa quantidade pode-se ser feita a divisão dos pontos. Observe a Figura 7.1 que mostra este passo.

Depois disto, é preciso encontrar os pontos de junção. Os pontos de junção são os pontos de quebra, já mencionados anteriormente, projetados na linha divisora. Para encontrar as junções no modelo CGM, considere a matriz c que contém todos os casamentos $c(i,j)'s$. Para cada j, todos os processadores calculam a maior classe entre os pontos de quebra daquele j e caso haja mais de um ponto de quebra nesta classe, escolha o ponto de menor i. Para isto, primeiramente é feito o cálculo local e guardado em um vetor de tamanho m. Todos os processadores enviam esse vetor para o processador de maior índice e esse faz o cálculo das junções localmente. Ou seja, esse processador calcula para cada j a maior classe recebida com menor i. A seguir para cada classe já calculada tome o ponto com o menor j desta classe como junção. A junção 0 é inserida à esquerda das demais junções nos processadores que calculam os pontos de junção. Assim têm-se os pontos de junção que são enviados

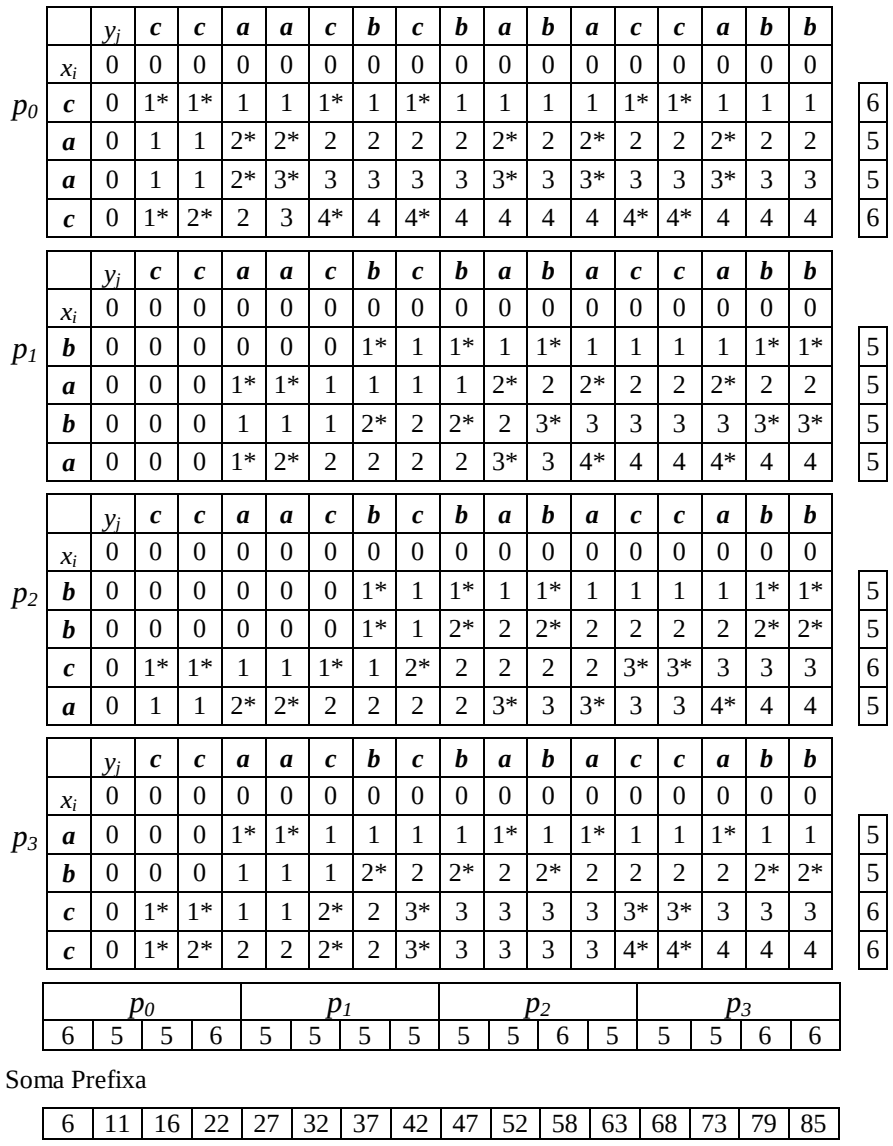


Figura 7.1: Distribuição das sequências e soma prefixa para encontrar os casamentos.

aos processadores para que os pontos que estejam na metade inferior recalculem suas classes a partir dos pontos de junção. Os pontos na metade superior mantêm suas classes inalteradas. Para que a divisão dos pontos ocorra de forma balanceada, é necessário dividir os pontos igualmente. A mesma quantidade na parte de cima também deve haver na parte debaixo. Geralmente para que essa condição seja satisfeita é preciso passar a linha horizontal divisora no meio dos pontos de um processador, dividindo os pontos igualmente em duas partes.

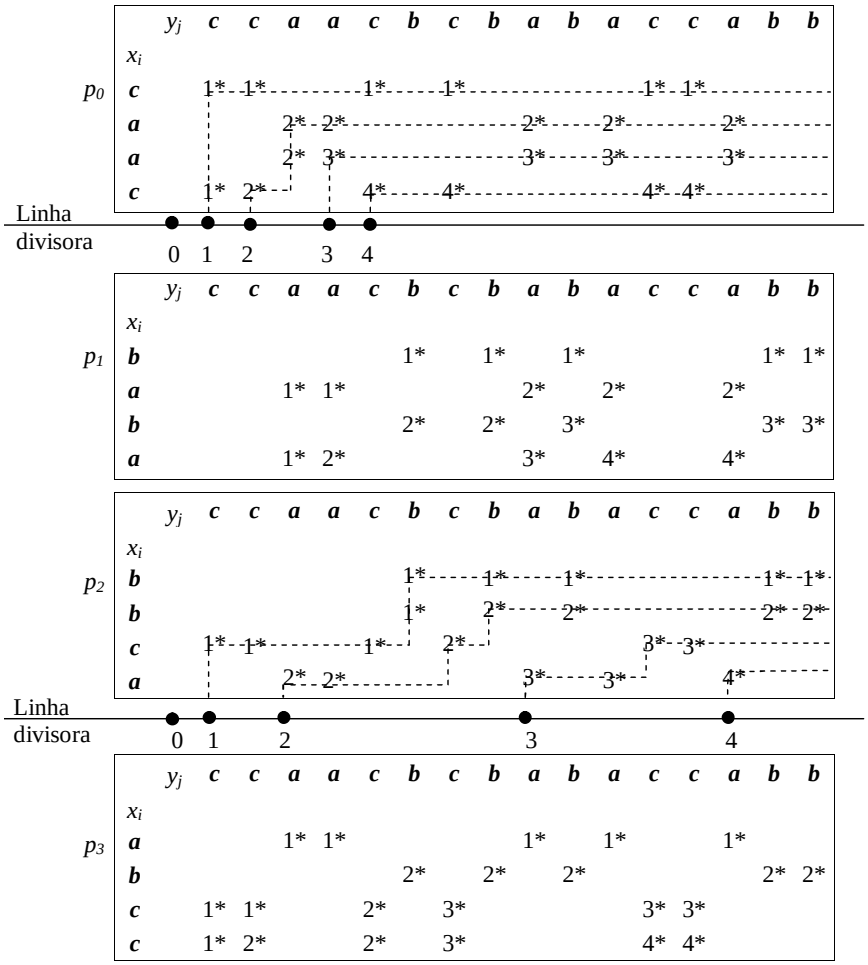


Figura 7.2: Pontos de junção na primeira rodada de comunicação.

Observe a Figura 7.2, o exemplo tem 4 processadores onde estão distribuídos todos os pontos. A primeira rodada de processamento para encontrar os pontos de

junção é feita nos processadores p_0 e p_2. Os pontos de junção encontrados em p_0 são projetados na linha divisora que está entre os processadores p_0 e p_1 e os pontos de junção encontrados em p_1 são projetados na outra linha divisora que está entre os processadores p_2 e p_3.

A Figura 7.3 demonstra a segunda rodada de processamento. Os pontos de junção encontrados estão nos processadores p_0 e p_1. Projetam-se esses pontos na linha divisora que está entre os processadores p_1 e p_2.

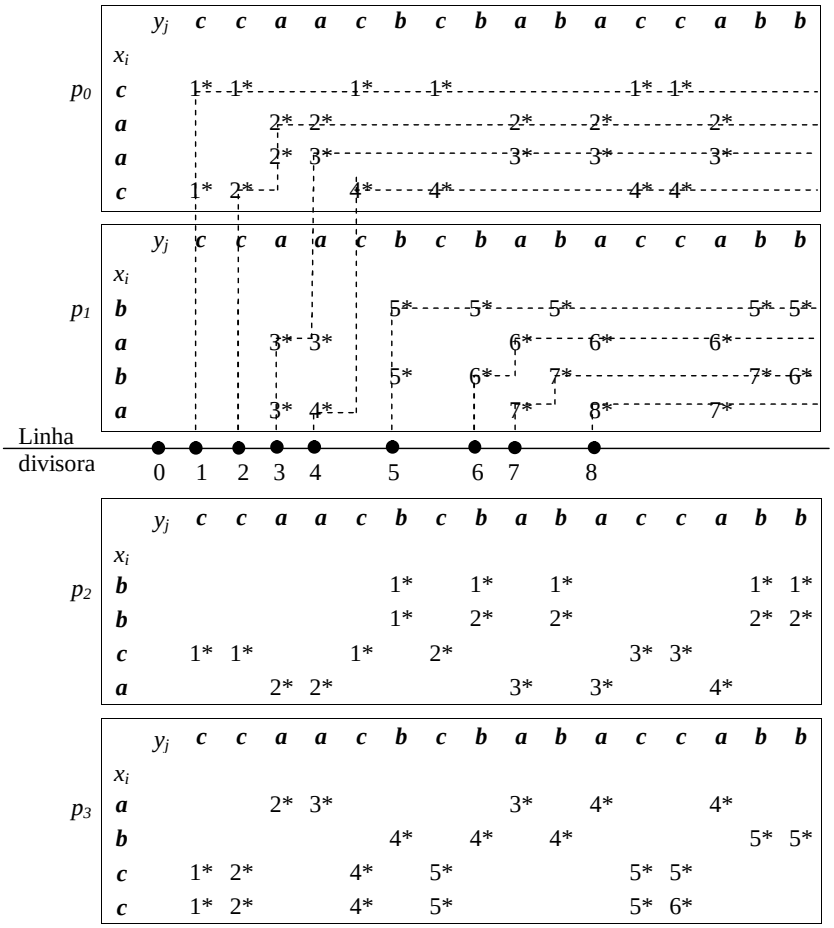


Figura 7.3: Pontos de junção na segunda rodada de comunicação.

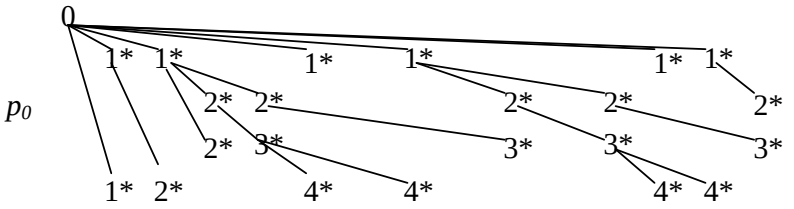
Até este momento a estrutura de dados utilizada é a matriz c com dimensão, $m \times n$, que está distribuída uniformente entre os processadores. Para dar continuidade ao algoritmo é necessário utilizar uma outra estrutura de dados. A estrutura de

dados mais adequada para esse algoritmo é uma árvore, onde cada nó pertencente a essa árvore representa um ponto. O conjunto de r pontos formado por todos os casamentos é usado para formar os r nós da árvore. Lembre-se que o ponto $f = (i_f, j_f)$ é pai do ponto $s = (i_s, j_s)$ se f domina s, ou seja, se $i_f < i_s$, $j_f < j_s$ e $classe(f) = classe(s) - 1$. A cada etapa concluída do algoritmo que gera uma modificação nos nós da árvore, deve haver um passo para atualizar os respectivos pontos na matriz c, sendo que cada nó terá os valores relevantes de cada ponto (i, j). Desta forma os pontos que estão armazenados tanto na matriz como na árvore estarão ao final de cada etapa coerentes entre si.

Para construir essa estrutura de árvore considera-se o conjunto que contém todos os pontos, conforme definido anteriormente. Inicialmente deve-se encontrar para cada ponto na matriz c o seu ponto pai. Isto é feito da seguinte forma: ordene o conjunto de pontos lexicograficamente pela $classe(p)$, por j e por i. Encontre o ponto f que é pai do ponto s da seguinte forma. O ponto s conhece sua posição (i, j) e sua classe, assim sendo ele procura no conjunto de pontos, o ponto f através de uma busca binária pela $classe(s) - 1$. Após encontrada esta classe fazer outra busca binária dentro desta classe agora pelo $j - 1$ e para finalizar a ordenação fazer outra busca binária desta classe deste j mas agora pelo $i - 1$. Isto resulta no ponto f da $classe(s) - 1$ tal que este ponto possui o maior j e o maior i que sejam menores do que o j e i de s. Observe a Figura 7.4 que mostra a execução deste passo.

Esse procedimento é feito por todos os pontos em todos os processadores paralelamente, formando uma floresta onde os pontos de junção são as raízes das árvores. Cada ponto da junção recebe no seu campo *flag* o valor 1. Depois de formada a floresta é necessário que todos os nós pertencentes a uma árvore saibam quem é o seu ponto da junção, ou seja, sua raiz. A esse passo já havia sido dado o nome de Atualização com Junção.

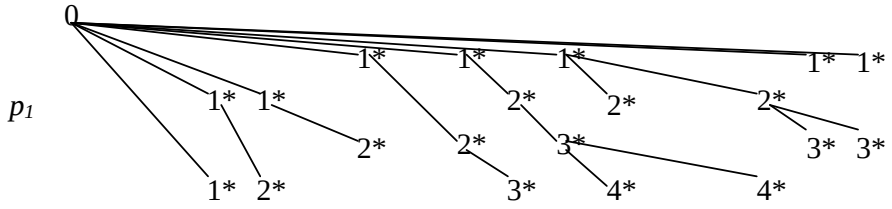
Na Atualização com Junção, calcule em cada nó da árvore a distância do nó até a raiz e armazene em uma variável na estrutura árvore, o valor da classe da raiz. Para calcular a distância entre o nó e a raiz de sua árvore, usa-se o *pointer jumping*. O *pointer jumping* é uma técnica que pode ser usada para determinar a profundidade de um nó em sua árvore [CMS01]. Para entender melhor o funcionamento do *pointer jumping*, considere uma árvore de altura 6. Veja a Figura 7.5 que demonstra o *pointer jumping* em uma árvore de altura 6. Na primeira rodada de comunicação, o nó i comunica-se com o nó $i + 1$ e armazena em $i + 1$ à distância de i somada a distância de $i + 1$. Na rodada seguinte o nó i comunica-se com o nó $i + 2$ e armazena em $i + 2$ à distância de i somada à distância de $i + 2$. Na última rodada o nó i comunica-se com o nó $i + 4$ e armazena em $i + 4$ a distância de i somada a distância de $i + 4$.



Classe 0	Classe 1	Classe 2	Classe 3	Classe 4
(i, j)	(i, j)	(i, j)	(i, j)	(i, j)
(0, 0)	(1, 1)	(4, 2)	(3, 4)	(4, 5)
	(4, 1)	(2, 3)	(3, 9)	(4, 7)
	(1, 2)	(3, 3)	(3, 11)	(4, 12)
	(1, 5)	(2, 4)	(3, 14)	(4, 13)
	(1, 7)	(2, 9)		
	(1, 12)	(2, 11)		
	(1, 13)	(2, 14)		

Ordenação

p_0



Classe 0	Classe 1	Classe 2	Classe 3	Classe 4
(i, j)	(i, j)	(i, j)	(i, j)	(i, j)
(0, 0)	(2, 3)	(4, 4)	(4, 9)	(4, 11)
	(4, 3)	(3, 6)	(3, 10)	(4, 14)
	(2, 4)	(3, 8)	(3, 15)	
	(1, 6)	(2, 9)	(3, 16)	
	(1, 8)	(2, 11)		
	(1, 10)	(2, 14)		
	(1, 15)			
	(1, 16)			

Ordenação

p_1

Figura 7.4: Árvores são formadas.

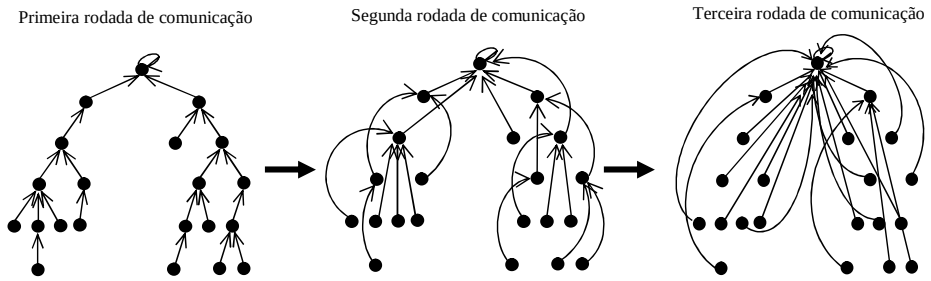


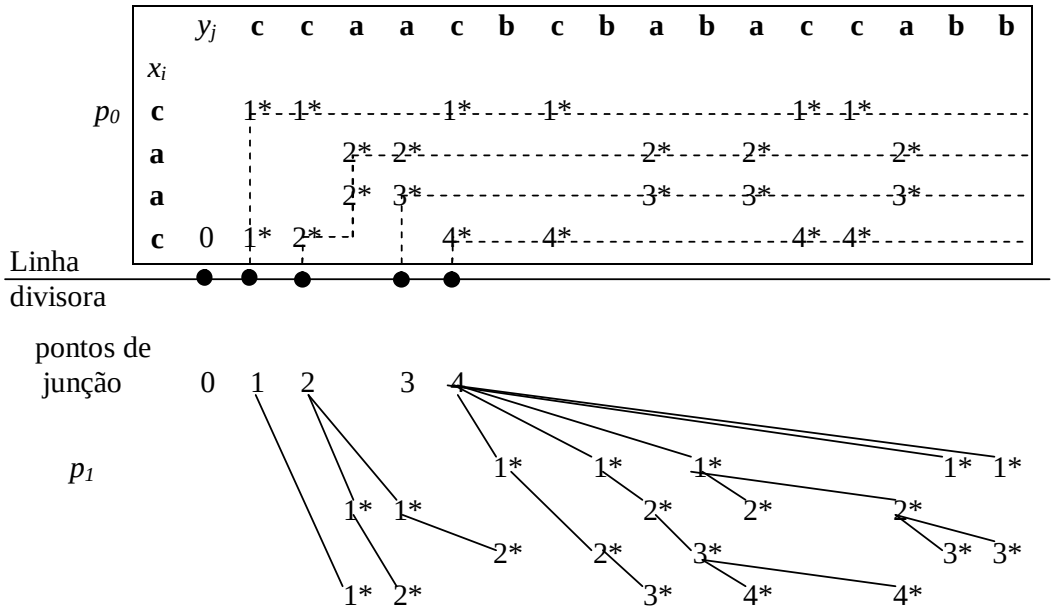
Figura 7.5: *Pointer jumping* em uma árvore de 6 nós.

Considere o conjunto de pontos para executar a Atualização com Junção. A raiz da árvore é o único nó que no início desse passo tem no campo *flag* o valor 1. O campo *flag* é usado no algoritmo como condição de parada para a Atualização com Junção. O nó s conhece o nó f, seu pai na árvore. Verifique se o campo *flag* de f é igual a 1, se verdade o nó f é a raiz da árvore, então o nó s calcula sua distância até a raiz, armazena o valor da classe da raiz no conjunto de pontos, recalcula sua classe somando-se a sua profundidade à classe da raiz e recebe o valor 1 em seu campo *flag*. Caso o campo *flag* do nó f não for igual a 1, o nó s verifica se o pai do pai (avó de s) tem o campo *flag* igual a 1, se verdade o avô do nó s é a raiz de sua árvore, sendo assim o nó s calcula sua distância até a raiz, armazena o valor da classe da raiz, recalcula sua classe somando-se a classe da raiz a sua profundidade e atribui a seu campo *flag* o valor 1. Ao final desse passo do algoritmo todos os nós da árvore conhecem a que árvore pertencem, suas profundidades na árvore e a classe da raiz desta árvore. Observe a Figura 7.6.

O próximo passo do algoritmo é a Atualização entre Árvores. Cada ponto nesse momento do algoritmo conhece a *classe* da raiz da árvore a qual ele pertence e está com sua classe atualizada até o momento. Para fazer a Atualização entre Árvores considere o conjunto de pontos, ordene lexicograficamente pela *classe* da raiz da árvore e por j.

Dado um ponto p, encontre um ponto nas árvores cujas raízes possuem classes maiores que as classes de p, ponto este que domina p, cuja classe poderá aumentar o valor da classe de p. Isto é feito de uma forma de árvore binária, onde na i-ésima iteração 2^i árvores são intercaladas juntas. Na iteração i denota-se por árvores direitas as 2^{i-1} árvores com raízes de maiores classes.

No conjunto de pontos, o nó $p = (i_p, j_p)$ é um nó da árvore esquerda com $classe(p)$, ele procura através de uma busca binária pela $classe(p) + 1$. Depois



Nesse passo cada nó é atualizado pelo ponto de junção.

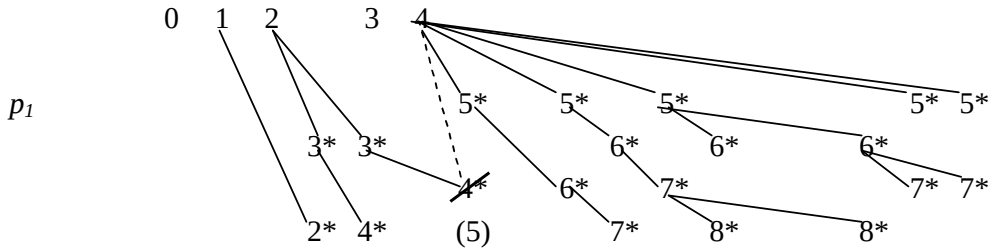


Figura 7.6: Atualização com Junção e entre Árvores.

de encontrada a $classe(p) + 1$, faz-se outra busca binária por um nó $t = (i_t, j_t)$ tal que $j_t < j_p$, porém que o j seja o de valor máximo. Quando o nó p encontra no conjunto de pontos um nó t na árvore direita, o nó p calcula a diferença da classe de seu pai com a classe do pai do nó t, isso é feito repetidamente em todos os nós. Se a diferença da classe do pai de p for maior que a diferença do pai de t, então ele permanece na mesma árvore. Se a diferença da classe do pai de p for menor que a diferença do pai de t então o nó p muda de pai e atualiza sua classe conforme a classe de seu novo pai, atualizando também a árvore a qual pertence nesse momento. Mesmo encontrado o ponto t, pode acontecer que este ponto não seja a melhor escolha, então ele continua procurando nas árvores direitas pelo ponto t, até que ele finalize essa busca. Somente será alterado o pai de p quando encontrar o ponto t pertencente às árvores da direita que aumente o valor de sua *classe*. Na Figura 7.6, a linha tracejada mostra que um ponto t foi encontrado à direita de um ponto p e a $classe(t)$ é maior que a $classe(p)$, então o ponto p mudou de pai e sua classe foi atualizada de 4 para 5.

Faz-se necessário analisar o tempo de execução após a descrição deste algoritmo.

Executando em p processadores, a complexidade de tempo do algoritmo é a seguinte como segue: Para encontrar o número total de casamentos usa-se a soma prefixa em cada linha, gasta-se tempo $O(\frac{nm}{p})$ com $O(1)$ rodadas de comunicação nesse passo. Para encontrar os pontos de junção em cada coluna usa-se a operação do máximo prefixo em cada coluna, com a mesma complexidade. Para construir a árvore usa-se a ordenação que gasta tempo $O(\frac{r}{p} \log r)$ com $O(1)$ rodadas de comunicação. Para realizar a Atualização com Junção faz-se o *pointer jumping* em cada árvore formada o que gasta tempo $O(\frac{nm}{p})$ e $O(\log p)$ rodadas de comunicação. A Atualização entre Árvore precisa de $O(\log p)$ rodadas para ser concluída, onde em cada rodada usa-se uma ordenação lexicográfica e uma busca binária. Portanto, como as junções da metade superior com a metade inferior são feitas duas a duas, precisa de $O(\log p)$ rodadas, totalizando $O(\log^2 p)$ rodadas de comunicação e tempo $O(nm/p)$ em cada rodada de computação.

Capítulo 8

Conclusão

O problema LCS é aplicável a várias situações como na correção ortográfica, na comparação de arquivos e em [SM97] encontra-se sua aplicação na comparação de sequências de DNA e no mapeamento de proteínas. Este problema é resolvido sequencialmente em tempo quadrático, para sequências de mesmo comprimento, através da técnica de programação dinâmica descrita em [CLRS02].

Nos dias atuais, a quantidade de dados disponíveis é enorme. Para que se possa utilizar todas essas informações é preciso ter um grande poder de processamento para estes dados. Técnicas de processamento paralelo têm sido aplicadas para resolver esse problema. Entretanto, através da literatura pode-se constatar que técnicas aplicadas são específicas para o modelo PRAM, que muitas vezes é difícil de adaptar para as máquinas realísticas. Em [BS97] e [Lu90] são apresentados algoritmos para resolver o problema LCS, porém não demonstram um bom desempenho e dependem da existência de uma memória compartilhada.

Após o estudo e a análise do algoritmo sequencial LCS, implementou-se um algoritmo simples para o modelo CGM com base nesse algoritmo, o qual apresentou um resultado insatisfatório. Pôde-se observar que para o problema LCS é necessário criar um algoritmo paralelo mais eficiente. Isto motivou o estudo de outros algoritmos paralelos para o modelo CGM.

Este trabalho teve como objetivo principal estudar a complexidade computacional apresentada em algoritmos paralelos no modelo CGM para resolver o problema LCS. O modelo CGM é um modelo realístico de computação paralela em que são considerados, além da complexidade de tempo de processamento, o custo de comunicação entre os processadores.

Num primeiro momento teve-se a intenção de implementar o algoritmo apresentado em [ACS03], estudou-se as estruturas de dados e a técnica desenvolvida no trabalho e pode-se observar claramente a dificuldade na implementação por se tratar de um algoritmo com estruturas de dados sofisticadas, apesar de ter $O(\log p)$ rodadas de comunicação. Optou-se em estudar outro algoritmo desenvolvido para o modelo PRAM apresentado por [Lu90].

Este algoritmo PRAM apresenta estruturas de dados mais simples, o estudo sobre esse algoritmo resultou num algoritmo para o modelo CGM, porém não implementado. A adaptação de técnicas usadas no PRAM para o CGM apresenta algumas dificuldades como por exemplo a ausência de uma memória compartilhada (o que pode implicar em replicação de dados e aumento do espaço requerido) e na necessidade de reduzir o número de rodadas de comunicação. Teoricamente, pode-se observar durante o estudo de adaptação do algoritmo do modelo PRAM para o modelo CGM um desempenho desfavorável, utilizando de muitas rodadas de comunicação para chegar ao resultado final.

Uma das contribuições deste trabalho é apresentar a complexidade computacional apresentada nos algoritmos paralelos para resolver o problema LCS no modelo CGM. Como trabalho futuro pode-se: (i) implementar o algoritmo CGM estudado no Capítulo 5; (ii) implementar o algoritmo proposto descrito no Capítulo 7 para o modelo CGM e analisar os resultados obtidos; (iii) estudar como melhorar as estruturas de dados usadas nos algoritmos dos modelos realísticos; (iv) pesquisar e analisar estratégias de como diminuir as rodadas de comunicação nesses algoritmos.

Referências Bibliográficas

[ACS03] C. E. R. Alves, E. N. Cáceres, e S. W. Song. A BSP/CGM Algorithm for the All-Substrings Longest Common Subsequence Problem. In *17th IEEE Annual International Parallel and Distributed Processing Symposium (IPDPS 2003)*, páginas 22–26. IEEE Computer Society, 2003.

[BS97] K. N. Babu e S. Saxena. Parallel algorithms for the longest common subsequence problem. In *Proceedings of the Fourth International Conference on High-Performance Computing*, páginas 1–6. IEEE Computer Society, 1997.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, e T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, páginas 1–12. 1993.

[CLRS02] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein. *Algoritmos: Teoria e Prática*. Editora Campus, 2002.

[CMS01] E. N. Cáceres, H. Mongelli, e S. W. Song. Algoritmos paralelos usando CGM/PVM/MPI: Uma Introdução. In *XXI Congresso da Sociedade Brasileira de Computação, Jornada de Atualização de Informática*, páginas 219–278. 2001.

[DFRC93] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computacional geometry for coarse grained multicomputers. In *Proceedings of the 9th Annual ACM Symposium on Computational Geometry*, páginas 298–307. 1993.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[Jáj92] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[Lu90] Mi Lu. Parallel computation of longest-common-subsequence. In *ICCI'90: Proceedings of the International Conference on Advances in Computing and Information*, páginas 385–394. Springer-Verlag New York, Inc., 1990.

[MMT95] B. M. Maggs, L. R. Matheson, e R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, volume 2, páginas 61–70. 1995.

[Rei93] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kauffman, 1993.

[Sch98] J. Schmidt. All highest scoring paths in weighted graphs and their application to finding all approximate repeats in strings. *SIAM J. Computing*, 27(4):972–992, 1998.

[SM97] J. Setubal e J. Meidanis. *Introduction to Computacional Molecular Biology*. PWS Publishing Company, 1997.

[Val90] L. C. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.