Alinhamento de Várias Sequências Utilizando Arquiteturas Paralelas Híbridas

Valter de Oliveira Ferlete

SERVIÇO DE PÓS-GRADUAÇÃO DA FACOM-UFMS

Data de Depósito:

Assinatura:_____

Valter de Oliveira Ferlete

Orientador: *Prof. Dr. Marco Aurélio Stefanes*

Dissertação apresentada à Faculdade de Computação da Universidade Federal de Mato Grosso do Sul como parte dos requisitos necessários à obtenção do título de Mestre em Ciência de Computação.

Banca Examinadora:

- Prof. Dr. Marco Aurélio Stefanes (Orientador) (FACOM / UFMS)
- Prof. Dr. Renato Porfirio Ishii (FACOM / UFMS)
- Prof. Dr. Luiz Carlos da Silva Rozante (UFABC)

UFMS - Campo Grande
Setembro/2015

Agradecimentos

Agradeço a Deus pela permissão e sabedoria em concluir este trabalho, pois ao escolher trilhar o caminho da sabedoria conheci a mulher da minha vida e futura esposa Tatiane de Queiroz Moura, que muito me apoiou na conclusão desta jornada.

Aos meus pais Antonio Ferlete e Idalíria de Oliveira Ferlete, por estarem sempre ao meu lado apoiando e torcendo pela realização deste sonho e futuramente realizar o sonho de minha mãe em ter um doutor na família.

Aos meus tios Fábio Assis Martins e Clarice Fellete, pela hospitalidade e preocupação durante o início desta caminhada e inúmeras outras.

Ao professor Marco Aurélio Stefanes, por todos os ensinamentos, conselhos e ajuda. Obrigado pelo tempo dedicado, sempre direcionando meus estudos e pesquisas.

A todos os professores da FACOM que contribuíram para meu aperfeiçoamento profissional e pessoal.

Aos amigos que ganhei durante o curso: Eduardo Machado Real, Francisco Sanches Filho, Kleber Kruger, Patrik Olã Bressan, Simone Araújo, Luiz Alvino, Jean Carlo Wai Keung Ma, Angelo Maggioni e Silva, Hudson Fujikawa de Paula e Rosemir Moreira; obrigado por me propiciarem momentos alegres e descontraídos. Isso com certeza fez minha caminhada até aqui mais fácil.

Abstract

The comparison of biological sequences is one of the main tools of bioinformatics to assist biologists to perform data analysis in order to determine the function or structure of biological sequences and infer information about their evolution in organisms that are being studied. The resolution of this problem, however, involves large computational and biological difficulties, leading to the emergence of various approximations and heuristics for its resolution. The goal of this work is to write a parallel algorithm in CUDA and MPI, to attain the overall alignment of multiple biological sequences, specifically DNA and proteins using heuristics that can provide a certain quality in a reasonable time. A comparison of the results was carried out with the data that today's best tools present.

Resumo

A comparação de sequências biológicas é uma das principais ferramentas da bioinformática, para auxiliar os biólogos a realizar análise de dados com objetivo de determinar a função ou estrutura das sequências biológicas e inferir informações sobre sua evolução em organismos que estejam em estudo. A resolução deste problema, contudo, envolve grandes dificuldades computacionais e biológicas, levando ao surgimento de diversas aproximações e heurísticas para sua resolução. O objetivo deste trabalho é escrever um algoritmo paralelo em CUDA e MPI, para realizar o alinhamento global de várias sequências biológicas, especificamente de DNA e proteínas, utilizando heurísticas que possam fornecer uma certa qualidade em um tempo razoável. Foi realizado um comparativo dos resultados obtidos, com os dados que as melhores ferramentas da atualidade apresentam.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

CAPÍTULO

1

Introdução

A bioinformática abrange muitos problemas relacionados à biologia que usam ferramentas computacionais, dentre os quais existe um muito explorado denominado comparação de sequências biológicas [dB03]. Este campo interdisciplinar visa o desenvolvimento de ferramentas e algoritmos para auxiliar os biólogos na análise de dados, cujo um dos objetivos é identificar regiões similares que possam ser consequência de relações funcionais, estruturais ou evolucionárias entre as sequências. Nesse contexto, tem-se um aumento do desenvolvimento de projetos que envolvem sequenciamento genético, levando o crescimento dos bancos de dados genômicos como Genbank [BCC+13], com mais de 380.000 organismos e transcriptomas, tornando-se necessário desenvolver ferramentas cada vez mais rápidas e eficientes para comparação e análise dessas sequências.

A comparação de sequências biológicas pode ocorrer entre um par de sequências ou entre várias sequências. Ao utilizar duas ou mais sequências propicia a visualização das diferenças e semelhanças entre as sequência como um todo. Sendo utilizado para caracterizar famílias de sequências, inferir a história evolucionária das sequências e buscar por sequências homologas em banco de dados entre outras aplicações [Gus93].

Encontrar o alinhamento ótimo de $k > 2$ sequências $S = (s_1, s_2, \ldots, s_k)$ é um problema de alta complexidade. Algoritmos de programação dinâmica garantem encontrar o melhor alinhamento de k sequências de tamanho médio n com tempo de execução O(n^k) [CL88]. Por isso, muitas soluções heurísticas foram propostas para este problema. O problema de alinhar várias sequências foi provado formalmente por Wang e Jiang [WJ94] ser NP-Completo utilizando soma de pares e MAX SNP-Difícil utilizando árvore. Bonizzoni e Della Vedova

[BDV01] resolve uma questão em aberto, mostrando que o problema é NP-Completo para casos muito restrito em que a pontuação é uma métrica.

Existem diversas soluções heurísticas para este problema, sendo que as mais representativas são o alinhamento progressivo e o alinhamento iterativo [Mou04]. Os métodos que usam heurísticas funcionam bem para casos práticos, porém não garantem uma solução próxima da ótima e são fundamentados em métodos não rigorosos, mesmo que forneçam resultados satisfatórios.

No alinhamento progressivo, as sequências de maior similaridade são alinhadas e então as demais sequências são adicionadas ao alinhamento. O alinhamento progressivo é um dos métodos mais utilizados atualmente devido ao seu bom desempenho, sendo que uma das ferramentas mais conhecidas baseada neste método é o Clustal W [THG94]. Contudo, este tipo de solução pode causar uma degeneração no resultado final devido a propagação de buracos (*gaps*) quando um escolha ruim for tomada nos passos iniciais do processo.

O alinhamento iterativo permite o realinhamento de sequências, buscando minimizar o impacto causado por erros iniciais no alinhamento, reduzindo a probabilidade de propagá-los para o alinhamento final. Existem diversas soluções iterativas para o alinhamento de várias sequências como por exemplo, o modelos ocultos de Markov [JEP10], algoritmos genéticos [NH96], entre outros. Este método em geral demanda mais tempo de processamento que o alinhamento progressivo, contudo, é capaz de efetuar buscas mais amplas no espaço de soluções.

As soluções exatas para o alinhamento de sequências, utilizam métodos de programação dinâmica, tais como a estratégia de Carrilo-Lipman [CL88], que consome muito tempo e espaço, mesmo para quantidades pequenas de sequências o que pode tornar a solução pouco prática.

Outra maneira de obter algoritmos eficientes para problemas difíceis é abandonar a otimalidade das soluções e manter a generalidade do problema e a exigência de tempo polinomial. Para estes casos são utilizados os *algoritmos de aproximação*. Os algoritmos de aproximação fornecem uma garantia que todas as soluções estejam próximas de uma solução ótima (por exemplo, no máximo 10% da solução ótima). Esta garantia é chamada de *razão de aproximação*.

Para que os resultados dos algoritmos de alinhamento de várias sequências sejam obtidos em menor tempo, plataformas de alto desempenho estão sendo exploradas. Dentre essas, destacam-se as unidade de processamento gráfico (*Grafic Processing Units* - GPU). As gerações atuais de GPUs tais como *GeForce* e *ATI Radeon*, contém centenas de núcleos de processamento [LR09], sendo

largamente utilizadas em problemas onde a abordagem paralela de dados é usada.

O paralelismo também pode ser obtido através da utilização de *clusters* de computadores, sendo que cada nó de processamento é um computador praticamente completo, com processador, memória e disco rígido. Os nós de processamento são interligados por uma rede de interconexão dedicada. Atualmente, uma das ferramentas mais utilizadas para explorar paralelismo em *cluster* é o padrão MPI (*Message Passing Interface*), que permite a troca de mensagens síncronas e assíncronas entre os processos que executam em computadores distintos.

O objetivo desta dissertação de mestrado é propor e avaliar uma solução paralela para o alinhamento de várias sequências de aminoácidos, com foco principal em métodos que utilizam heurísticas e realizam o alinhamento progressivo, utilizando um *cluster* com soluções paralelas híbridas de computação em GPU e MPI. O alinhamento entre pares de sequências é realizado na GPU utilizando CUDA e a distribuição das tarefas entre os nós de processamento é realizada pelo MPI. Escolheu-se o algoritmo LzyRScan-mNW [TLS+14] que explora o acesso a memória compartilhada, realizando a divisão da matriz de alinhamento em fatias verticais de k colunas. Foi proposta esta solução devido ao seu alto paralelismo potencial. No segundo estágio, a criação da árvore guia é realizada pelo método Neighbor-Joining [SN87] utilizando apenas uma GPU. No último estágio é realizado o alinhamento progressivo utilizando MPI e GPU, seguindo o algoritmo do Clustal [HS88]. Ao final da dissertação, uma comparação do algoritmos ClustalW-MPI [Li03] com a implementação proposta e os fatores que influenciaram o desempenho da estratégia em GPU e MPI são avaliados.

O restante dessa dissertação está organizado como descrito a seguir. No Capítulo 2 são apresentados os conceitos básicos para entendimento e desenvolvimento do trabalho proposto. No Capítulo 3 são abordados os algoritmos sequenciais exatos, aproximadas e heurísticas para alinhamentos de várias sequências. No Capítulo 4 abordam-se as arquiteturas paralelas e como elas podem acelerar a resolução de problemas. No Capítulo 5 são apresentados alguns algoritmos paralelos utilizados para solução do alinhamento de várias sequências. No Capítulo 6 é realizado o detalhamento da solução híbrida proposta para o alinhamento de várias sequências, utilizando GPU e MPI. No Capítulo 7 são apresentado os resultados obtidos pela implementação proposta e realizado uma comparação com o ClustalW-MPI e no Capítulo 8 a conclusão é apresentada com sugestões para trabalhos futuros.

CAPÍTULO

2

Conceitos básicos

Este capítulo apresenta as definições necessárias para a compreensão e desenvolvimento do trabalho proposto. É realizada uma breve descrição dos conceitos biológicos, alfabeto, símbolos e sequências. Em seguida é definido o problema de alinhar duas sequências e também descrito os tipos de alinhamentos, distância de edição e similaridade, métricas, árvores filogenéticas, matrizes de pontuação e substituição. Por fim, é apresentado o tópico principal de estudo deste trabalho, o alinhamento de várias sequências.

2.1 Conceitos biológicos

Todo organismo vivo é caracterizado pela capacidade de se reproduzir e desenvolver. O Genoma de um organismo é definido como uma coleção de DNA (do inglês *Deoxyribonucleic acid*, ácido desoxirribonucleico), dentro do organismo, incluindo um conjunto de genes que codifica uma proteína [Pev05].

O DNA foi descoberto em 1869, quando o material hereditário foi isolado pela primeira vez [Dah05], período este que muitos conceitos em biologia foram estabelecidos. Embora a maioria destas descobertas e conceitos fosse recebida com grande interesse pela comunidade científica na época, a descoberta do DNA foi subestimada. Apesar de revelar a base molecular da vida celular, tornou-se um dos problemas fundamentais para a biologia.

James Watson e Francis Crick mostraram que em células vivas, duas cadeias de DNA estão interligadas formando uma dupla hélice conforme mostra a Figura 2.1 [Bro06].

O DNA é composto por polímeros não ramificados em que as subunidades monoméricas são quatro nucleotídeos [Bro06] quimicamente distintos: as

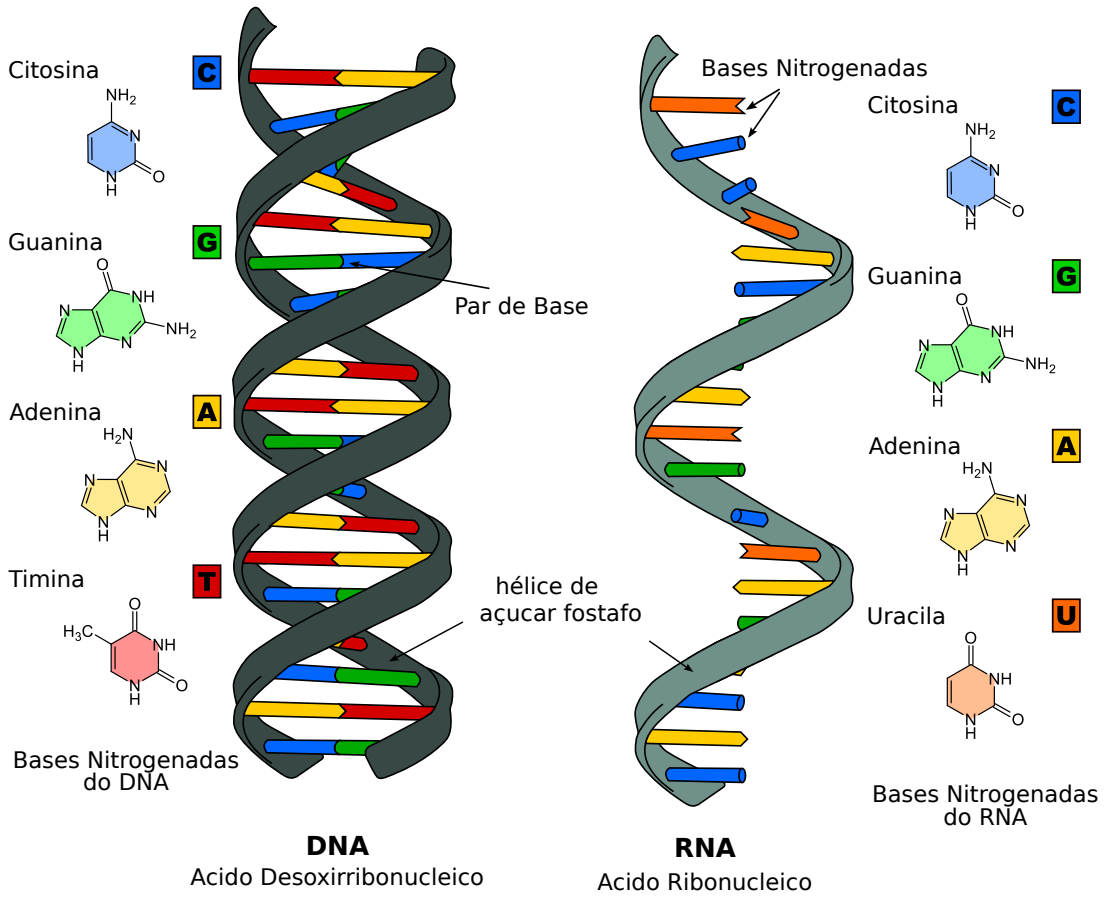


Figura 2.1: Representações gráfica das diferenças entre o DNA e RNA. No DNA um par de base A-T com duas pontes de hidrogênio e um par de base G-C com três pontes de hidrogênio. O RNA com uma base Uracila. [Bro06]

purinas Adenina (representada pela letra A) e Guanina (G), e as pirimidinas Timina (T) e Citosina (C). O grupo hidroxila do carbono-3 da pentose do primeiro nucleotídeo se une ao grupo fosfato ligado à hidroxila do carbono-5 da pentose do segundo nucleotídeo através da ligação fosfodiéster. Desta forma, os nucleotídeos se unem, constituindo uma fita de ácido desoxirribonucleico conforme é mostrado na Figura 2.2.

O RNA (do inglês *Ribonucleic acid*, ácido ribonucleico), é um polinucleotído similar ao DNA, com duas diferenças químicas importantes. Primeiro o açúcar no RNA é uma ribose e segundo o RNA contém mais uma base, a uracila representado pela letra U [Bro06]. O RNA é o responsável pela síntese de proteínas das células. Uma das principais áreas de pesquisa biológica hoje é como estas proteínas são construídas somente com 20 aminoácidos diferentes [LBZ+00]. Os aminoácidos podem ser classificados em algumas categorias distintas baseados principalmente na sua solubilidade em água, que é influenciado pela polaridade das suas cadeias. Na Tabela 2.1 é listado os aminoácidos existentes.

Neste trabalho de dissertação serão utilizados os aminoácidos para realizar o alinhamento de várias sequências.

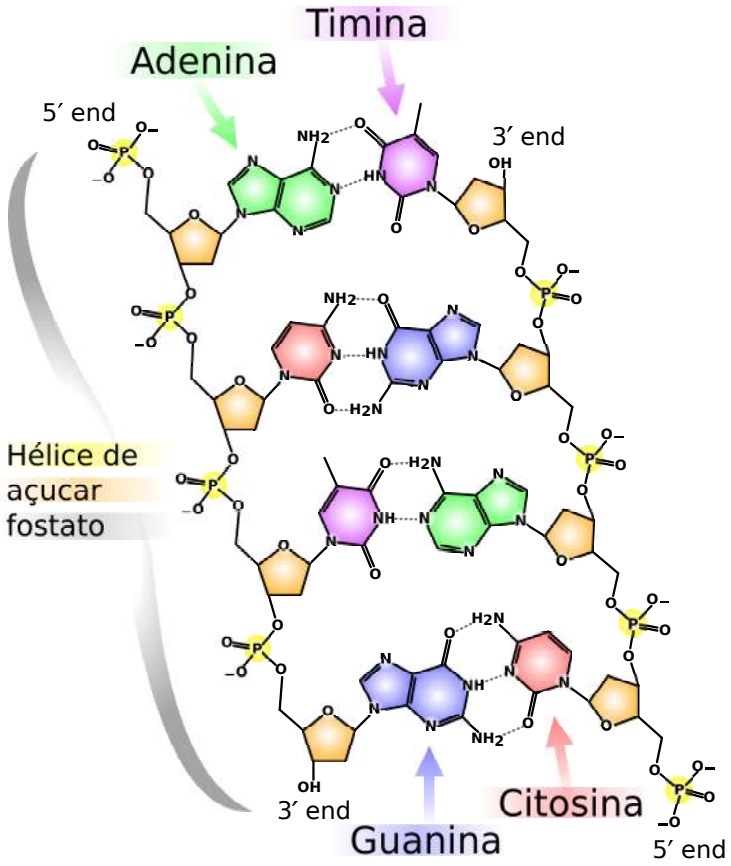


Figura 2.2: Representação química da ligação entre os nucleotídeos na dupla hélice do DNA. [Bro06]

2.2 Alfabeto, Símbolos e Sequências

Um *alfabeto* Σ é um conjunto finito e não-vazio de *símbolos*. Quando se refere a alinhamento de sequências genômicas, o *alfabeto* Σ é correspondente às bases nitrogenadas (ou simplesmente bases) presente em sequências de DNA, ou seja, $\Sigma = \{A,\ C,\ G,\ T\}$, e para os aminoácidos $\Sigma = \{A,\ C,\ D,\ E,\ F,\ G,\ H,\ I,\ K,\ L,\ M,\ N,\ P,\ Q,\ R,\ S,\ T,\ V,\ W,\ Y\}$. Uma *sequência* finita *s* sobre o alfabeto Σ é uma sequência *s*=(σ_1,σ_2, ..., σ_n) $\in \Sigma^n$, onde *n* $\geq$ 0 é um inteiro. Diz-se que o *comprimento* de *s*, denotado por $|s|$, é *n*. A sequência a^n é formada por *n* símbolos a.

2.3 Alinhamento

Sejam *k* um inteiro positivo e $s_1, s_2, \ldots, s_k$ sequências sobre um alfabeto Σ, com $|s_i|$ = n_i, para i=1,...,k. Um *alinhamento* A de $s_1, s_2, \ldots, s_k$ é uma matriz $A = (A_{ij})$ de dimensões $k \times n$ com entradas em Σ' tal que, para cada i, existe um conjunto J_i = {j_1,j_2,...,j_{n_i}} $\subseteq$ {1,2,...,n }, com $j_1 < j_2 < \ldots < j_{n_i}$ e tal que $A_{ij_1}\ A_{ij_2}\ \ldots A_{ij_{ni}}$ = s_i e tal que para todo $j \in$ {1,2,...,n } - J_i, temos A_{ij} = ␣. Chama-se *espaço* o símbolo ␣. Este símbolo é utilizado para representar uma

Aminoácido	Símbolo	Abreviação
Glicina ou Glicocola	Gly, Gli	G
Alanina	Ala	A
Leucina	Leu	L
Valina	Val	V
Isoleucina	Ile	I
Prolina	Pro	P
Fenilalanina	Phe ou Fen	F
Serina	Ser	S
Treonina	Thr, The	T
Cisteina	Cys, Cis	C
Tirosina	Tyr, Tir	Y
Asparagina	Asn	N
Glutamina	Gln	Q
Aspartato ou Ácido aspártico	Asp	D
Glutamato ou Ácido glutâmico	Glu	E
Arginina	Arg	R
Lisina	Lys, Lis	K
Histidina	His	H
Triptofano	Trp, Tri	W
Metionina	Met	M

Tabela 2.1: Listagem de Aminoácidos [Bro06]

inserção ou remoção em operações de edição.

Diz-se que dois caracteres $s_1[i]$ e $s_2[j]$ estão alinhados em A se $s_1[i]$ e $s_2[j]$ estão na mesma coluna de A.

Seja $A = (A_{ij})$ um alinhamento de s_1, s_2, ..., s_k. Um espaço de s_i no alinhamento A é qualquer segmento não-vazio maximal de A_i composto apenas de espaços, isto é, um espaço de s_i em A é um segmento $A_i[j..j']$ de A_i tal que:

- $j \leq j'$;
- todos os símbolos de $A_i[j..j']$ são $\llcorner$;
- não é possível estender $A_i[j..j']$ em A_i por meio de espaços em branco, ou seja, $j = 1$ ou $A_i[j\text{-}1] \neq \llcorner$ e $j' = n$ ou $A_i[j\text{+}1] \neq \llcorner$.

2.4 Alinhamento de duas sequências

De maneira informal um alinhamento de duas sequências s e t é obtido inserindo-se espaços nas sequências e então colocando-se uma sobre a outra de tal forma que cada carácter ou espaço esteja emparelhado a um único carácter (ou a um espaço) da outra cadeia. O uso de programação dinâmica para comparação de duas sequências iniciou-se com Needleman e Wunsch [NW70], para encontrar o alinhamento ótimo entre duas sequências. O algoritmo de

Needleman e Wunsch maximiza a pontuação entre duas sequências com um número mínimo de operações de inserção e exclusão, sendo conhecido como critério da máxima similaridade. O algoritmo de Seller [SS73] atribui um peso não negativo para substituições, inserções e exclusões. Este alinhamento, ao contrário de Needleman e Wunsch, minimiza a pontuação entre duas sequências, sendo conhecido como critério de mínima distância.

Edgar [ES04] distingue três classes de algoritmos para alinhamento de duas sequências. O método Sequência-Sequência, realiza sempre o alinhamento de somente duas sequências. Método Perfil-Sequência, utiliza o alinhamento de uma sequência com um conjunto de sequências, representado por um perfil e recentemente o método Perfil-Perfil tem sido definido como a construção do alinhamento de dois perfis.

Algoritmos baseados em programação dinâmica geralmente gastam tempo de computação da ordem de $O(nm)$, onde m e n corresponde ao tamanho das duas sequências a serem alinhadas. Este método utiliza uma matriz de pontuação que será abordado na seção 2.7.

Para simples ilustração, foi utilizado um esquema de pontuação entre as sequências, onde os símbolos iguais soma 1, e para símbolos diferentes subtrai 1 e espaços serão ignorados, pode-se observar que, para o alinhamento de duas sequências $s = \{ATGGCGT\}$ e $t = \{ATGAGT\}$, ao buscar uma similaridade entre essas sequências, um possível alinhamento conforme mostrado na Figura 2.3 a) tem-se uma pontuação total igual a 4, sendo que em b) obtém-se uma pontuação total igual a 2, logo conclui-se que o primeiro alinhamento é melhor que o segundo.

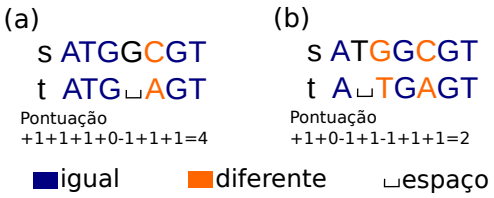


Figura 2.3: a) Melhor alinhamento entre $s = \{ATGGCGT\}$ e $t = \{ATGAGT\}$. b) um possível alinhamento entre $s = \{ATGGCGT\}$ e $t = \{ATGAGT\}$

2.5 Tipos de alinhamento

Mostra-se na Figura 2.4 os tipos de alinhamentos entre duas sequências. Os alinhamentos chamados *alinhamentos globais* Figura 2.4 a), procuram comparar as sequências de entrada s e t como um todo, encontrando a pontuação máxima entre elas. Este tipo de alinhamento é desejável em situações onde as sequências são similares, por exemplo, ao se alinhar genes ou proteínas homólogas.

Outro tipo de alinhamento também utilizado é o *alinhamento semi-global* como mostra a Figura 2.4 b). Este tipo de alinhamento é desejável em casos de montagem de genomas, onde se busca um alinhamento de pontuação máxima entre o prefixo de uma sequência e o sufixo de outra sequência, sem penalizar espaços em branco nas pontas das sequências.

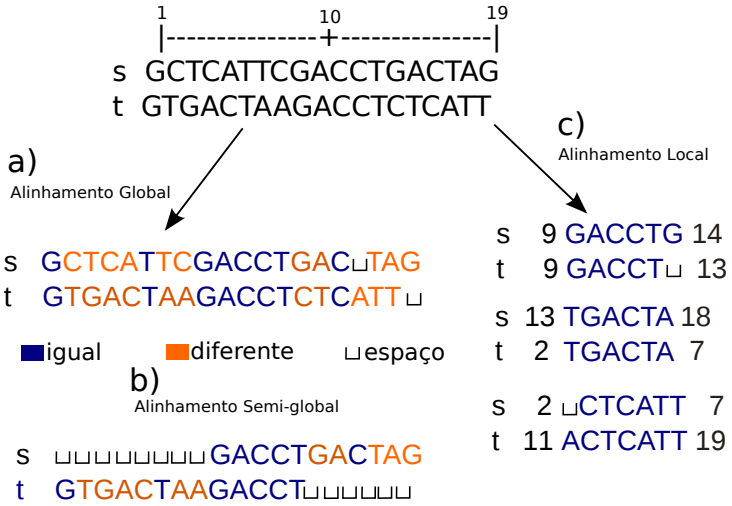


Figura 2.4: Tipos de alinhamentos entre sequências. a) Alinhamento Global. b) Alinhamento Semi-global. c) Alinhamento Local.

De forma semelhante ao alinhamento semi-global, pode-se estar interessados em comparar sequências s e t que globalmente possam não ser muito parecidas, mas que possuam trechos internos (não necessariamente prefixos ou sufixos) que tenham similaridade alta. Este tipo de alinhamento é conhecido como *alinhamento local*, como mostra a Figura 2.4 c), sendo desejável para se identificar trechos altamente conservados entre dois genomas.

2.6 Distância de edição e similaridade

Geralmente busca-se encontrar uma medida, seja diferença ou distância entre duas sequências biológicas, por exemplo, estrutural ou evolucionária. Dado um problema biológico esta medida pode ser diferente, onde se pode buscar por similaridade, a qual existe uma relevância prática maior, porém sem garantia formal, ou necessitar encontrar a distância de edição, a qual existe uma garantia de otimalidade. Esta medida é representada por uma matriz de pontuação.

Uma medida bem conhecida, chamada de distância de edição ou distância de Levenshtein [Lev66], está focada na transformação (edição) de uma sequência em outra, utilizando operações de inclusão, exclusão ou substituição de um caractere em uma única sequência. Este tipo de medida é utilizado em problemas de minimização quando é necessário realizar o alinhamento entre sequências biológicas, isto é, quando é necessário realizar a menor quantidade

de operações de edição para transformar uma sequência biológica em outra. Esta medida caracteriza o problema como de minimização, ou seja, realizar a quantidade mínima de operações para transformar uma sequência em outra.

Ao invés de encontrar a menor quantidade de operações para transformar uma sequência em outra, é possível comparar duas sequências de modo, a saber, o quão igual elas são. Neste caso busca-se encontrar a similaridade entre estas sequências. Para isto é utilizado uma maior pontuação quando dois símbolos de uma sequências são iguais e uma menor pontuação quando os símbolos são diferentes e uma penalização quando existe um espaço no alinhamento. Este tipo de medida é utilizado em problemas de maximização, ou seja, quando é necessário saber qual a pontuação máxima que uma sequência possui em relação à outra.

2.7 Matriz de pontuação

Ao realizar um alinhamento entre duas sequências s e t é possível atribuir um determinado valor quando os símbolos são iguais ou diferentes. Esta pontuação geralmente é representada por uma matriz, também conhecida como *matriz de pontuação*.

Seja $\Sigma_{\text{␣}} = \Sigma \cup \{\text{␣}\}$, uma matriz de pontuação é uma matriz que possui números reais como entrada e que são indexadas nas linhas e nas colunas por elementos de $\Sigma_{\text{␣}}$. Para $a, b \in \Sigma_{\text{␣}}$ e uma matriz de pontuação γ, denotamos $\gamma_{a \to b}$ a entrada de γ na linha a e coluna b. O valor de $\gamma_{a \to b}$ define o custo para a operação de substituição se $a, b \in \Sigma$, de inserção se $a = \text{␣}$, e de remoção se $b = \text{␣}$. A entrada $\gamma_{\text{␣}} \to \text{␣}$ não é definida. As operações de substituição, inserção e remoção sobre uma sequência são chamados de *operação de edição*.

Utilizando a matriz de pontuação p, define-se a pontuação ou custo $c_p(A)$ do alinhamento A entre s e t como

$$c_p(A) = \sum_{j=1}^{l} p(s'[j], t'[j])$$

onde p é a pontuação entre cada par de símbolo (a,b) de um alfabeto $\Sigma_{\text{␣}}$.

A Figura 2.5 mostra um exemplo de matriz de pontuação, utilizado para encontrar a distância entre duas sequências, onde o custo para alinhar símbolos iguais é 1, e o custo para alinhar símbolos diferentes é 0. O custo para substituir um símbolo por um espaço, ou um espaço por um símbolo é 0.

Este critério de pontuação entre pares de sequências, é conhecido também como pontuação - SP (*Sum-of-Pair*), utilizado para calcular a pontuação total do alinhamento.

γ=

	a	b	␣
a	1	0	0
b	0	1	0
␣	0	0	

Figura 2.5: Exemplo de matriz de pontuação.

2.8 Matriz de substituição

Métodos para alinhamento de sequências de proteínas normalmente medem similaridade utilizando uma matriz de substituição, com a pontuação para todas as possível trocas de uma aminoácido por outro, sendo a matriz PAM (do inglês, Percent Accepted Mutation, percentual de mutação aceito) e BLOSUM (do inglês, Block Substitution Matrice, matriz de substituição de blocos) [HH92], as mais relevantes.

As matrizes BLOSUM, são criadas a partir de alinhamentos locais de sequências de proteínas, com base no mínimo percentual de identidade do alinhamento. Por exemplo, a matriz BLOSUM62 exibe pelo menos 62% de alinhamentos idênticos.

Os valores de uma matriz BLOSUM são calculados conforme Equação 2.1, sendo p_{ij} a probabilidade de dois aminoácidos i e j estarem se substituindo em uma sequência homóloga, e f_i e f_j são as probabilidades de encontrar os aminoácidos i e j em qualquer sequência de proteína de forma aleatória. O Fator λ é um fator escalar, definido de tal forma que a matriz contenha valores inteiros [Zom06].

$$B_{ij} = \left(\frac{1}{\lambda}\right) \log\left(\frac{p_{ij}}{f_i * f_j}\right) \tag{2.1}$$

2.9 Métricas

Para realizar o alinhamento de várias sequências com um conjunto de operações de custo mínimo é necessário utilizar uma matriz de pontuação que induz uma métrica. Para um dado conjunto S, a função f: S x S $\rightarrow \mathbb{R}$ é uma *métrica* se f satisfaz simultaneamente

1. $f(s,s)$=0 (*reflexibilidade*),
2. $f(s,t) \geq 0$ (*não-negatividade*),
3. $f(s,t) > 0$ se $s \neq t$ (*positividade*),
4. $f(s,t) = f(t,s)$ (*simetria*) e

5. $f(s,u) \leq f(s,t) + f(t,u)$ (*desigualdade triangular*)

para cada $s,t,u \in S$.

Existem ainda outras classes de funções, onde nem todas as propriedades acima são satisfeitas. Por exemplo, para dado um conjunto S, é dito que f é uma *pramétrica* se $f(s,s)$=0 e $f(s,t) \geq 0$ para cada $s,t \in S$.

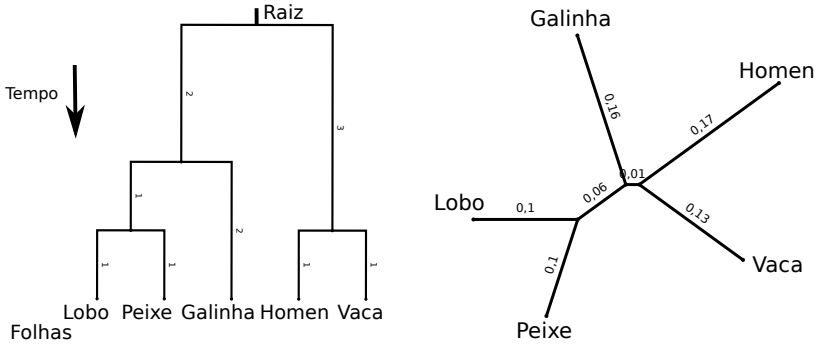
A Tabela 2.2 mostra, além de *métrica* (M), as propriedades de algumas classes: *pramétrica* (Pr), *semimétrica* (S), *hemimétrica* (H), *pseudométrica* (P) e *quasimétrica* (Q). Cada célula marcada com um "x" na linha i representa que a função da classe de função j requer, por definição, que a propriedade i seja satisfeita.

Propriedade \ Função	Pr	S	H	P	Q	M
$f(s,s)$=0	x	x	x	x	x	x
$f(s,t) \geq 0$	x	x	x	x	x	x
$f(s,t) > 0$ para $s \neq t$		x			x	x
$f(s,t) = f(t,s)$		x		x		x
$f(s,u) \leq f(s,t) + f(t,u)$			x	x	x	x

Tabela 2.2: Classe de funções satisfeita por propriedade. [Ara12]

2.10 Árvores Filogenéticas

Os estudos da similaridade em organismos sugerem fortemente que todo organismo na terra tem um ancestral em comum. Assim qualquer conjunto de espécies está relacionado e este relacionamento é chamado de filogenia. Normalmente este relacionamento pode ser representado por uma árvore filogenética. A tarefa da filogenética é inferir desta árvore observações sobre organismos existentes.



(a) Árvore com Raiz **(b)** Árvore sem Raiz

Figura 2.6: a) Exemplo de uma árvore binária, mostrando a raiz e as folhas e a direção do tempo evolucionário(o tempo mais recente inicia na parte debaixo da Figura). b) A árvore sem raiz correspondente. A direção de evolução é indeterminado. [DEKM98]

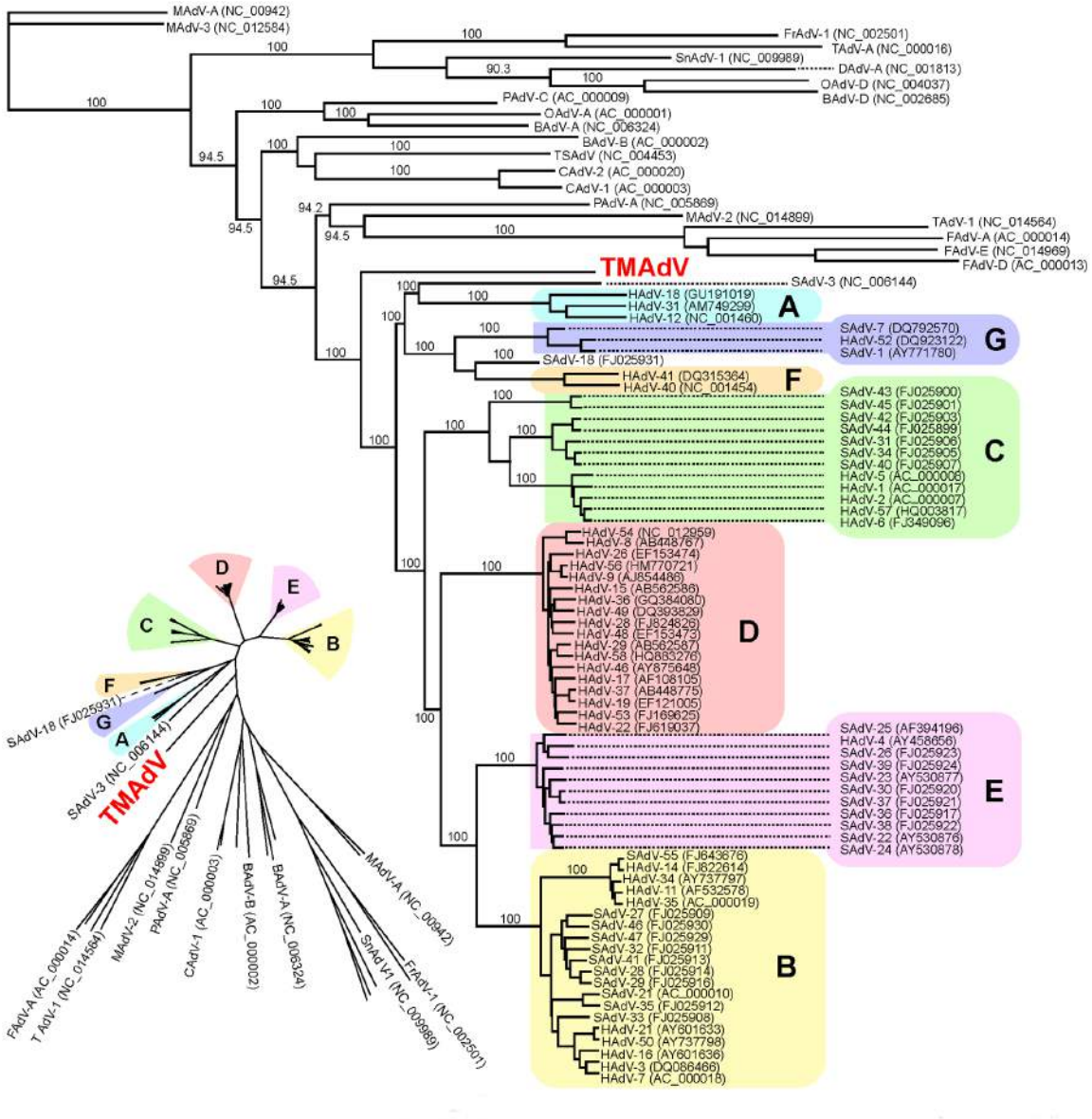


Figura 2.7: Reconstrução da árvore filogenética do adenovírus TMAdV totalmente sequenciados no GenBank. [CYK+11]

A Figura 2.6 ilustra dois tipos de árvore filogenética de algumas espécies. Nessa árvore as sequências que representam cada OTU (do inglês, *Operational Taxonomic Unit*, unidade taxonômica operacional) rotulam suas folhas. Cada aresta da árvore possui certa quantidade de divergência associada, definindo alguma medida de distância entre as sequências ou um modelo sobre o curso da evolução.

A verdadeira árvore filogenética possui uma raiz ou um último ancestral de todas as sequências, conforme sugere a Figura 2.7. Alguns algoritmos informam ou no mínimo conjecturam sobre a localização desta raiz. É possível observar algoritmos de alinhamento progressivo fazendo uso de uma "árvore guia" para guiar o alinhamento final.

Alguns métodos para construção da árvore como NJ (do inglês, *Neighbor-Joining*, Junção de Vizinho) e UPGMA (do inglês *Unweighted Pair Group Method*

with Arithmetic Mean, método de agrupamento de pares não ponderado com média aritmética) utilizam como entrada uma matriz de distâncias e devolvem uma árvore sem raiz.

2.11 Alinhamento de várias sequências

Principal problema estudado nessa dissertação, o alinhamento de várias sequências é considerado do ponto de vista da matemática uma extensão natural do alinhamento de pares de sequências. Abaixo é apresentada a definição do problema de alinhar várias sequências.

Problema 1 *(Alinhamento de Várias Sequências). Dado um inteiro $k \geq 2$ e k sequências $s_1, \ldots, s_k$ sobre um alfabeto Σ e fixada uma função de pontuação $\gamma = \Sigma_{\sqcup} \times \Sigma_{\sqcup} \to \mathbb{R} \geq 0$, encontrar um alinhamento A cujo custo $c_p(A)$ seja igual a $c(s_1, \ldots, s_k)$.*

Um alinhamento A de $s_1, \ldots, s_k$, cuja pontuação $c_p(A)$ seja igual a $c_p(s_1, \ldots, s_k)$ é dito um *alinhamento ótimo*.

Um exemplo do alinhamento de várias sequências é mostrado na Figura 2.8. Este alinhamento é composto com trechos de sequências de nucleotídeos do vírus HIV-1. Assim como no alinhamento de pares de sequências descrito na seção 2.4, é atribuído uma pontuação (ou custo) a cada alinhamento, sendo este parâmetro usando para decidir quais alinhamentos são os melhores.

```
CD014379.1|CD01     CTGTGCATAGAGGGTCTGACGGTCCTGTAACAGTGCCTG-AAAAATTAAC
FD200363.1|FD2      CAGCTGGTTCAATCCTCACTAAGGCACTTCAACAAAAACCACACCCTCAA
FD282571.1|FD2      CAGCTGGTTCAATCCTCACTAAGGCACTTCAACAAAAACCACACCCTCAA
FD295731.1|FD2      CAGCTGGTTCAATCCTCACTAAGGCACTTCAACAAAAACCACACCCTCAA
EH001789.1|EH0      CAAGGCTACGATGTTGGATTCTG--TTGTCTGGG---GACG-ACTAAAGC
BG234530.1|BG23     AATGGTCGTCATTTCCTCCAAGTAAATGGCATGATAAACAAAAAAAGCAC
```

Figura 2.8: Alinhamento de várias sequências. As sequências são trechos de nucleotídeos do vírus HIV-1 de organismos estudados. Os dados foram obtidos do banco de dados mundial chamado GenBank, disponível em http://www.ncbi.nlm.nih.gov. O alinhamento foi construído pelo Clustal W, versão 1.82.

Muitos métodos diferentes são propostos para resolução deste problema tais como: métodos progressivo, programação dinâmica, alinhamento de soma de pares (SP), alinhamento consenso e o alinhamento de árvore, dentre outros [CWC92]. O alinhamento de várias sequências tem características de otimização combinatória, sendo um problema NP-Difícil [WJ94], [BDV01], portanto uma solução exata se torna inviável para grandes quantidades de sequências, sendo possível utilizar duas abordagens, tais como heurísticas e algoritmos de aproximação.

Estes métodos de comparação de várias sequências possuem tempo de complexidade variando de $O(nl^2)$ até $O(l^n)$, onde n é o numero de sequências e l o tamanho das sequências [CWC92].

A utilização de programação dinâmica é uma maneira simples de calcular alinhamento de várias sequências, no entanto o custo desta abordagem é caro em termos de tempo de computação e espaço de memória.

Heurísticas como o alinhamento progressivo, têm sido utilizadas para superar estas restrições, aliadas a soluções paralelas devem ser vistos como alternativa para estes ou, pelo menos, como possíveis refinadores de uma solução ótima.

O foco deste trabalho foi direcionado a utilização de métodos de alinhamento progressivo, que basicamente realizam três passos para construção do alinhamento de várias sequências. Primeiro é construída a matriz de distâncias, utilizando programação dinâmica para realizar o alinhamento ótimo de pares de sequência. No Segundo passo é criada uma árvore guia com a matriz de distâncias e no terceiro passo é realizado o alinhamento progressivo utilizando a árvore guia criada anteriormente.

CAPÍTULO

3

Algoritmos sequenciais para alinhamento de sequências

Este capítulo apresenta algumas propostas para execução do alinhamento de sequências utilizando algoritmos sequenciais exatos e aproximados. Será abordado o método de Carrilo e Lipman como proposta para o alinhamento exato de várias sequências, o método de Needleman e Wunsch e o algoritmo de Smith Waterman para o alinhamento exato de pares de sequência. Também será apresentado o algoritmo aproximado proposto por Gusfield para encontrar o alinhamento de várias sequências.

3.1 Algoritmos exatos

Do ponto de vista computacional, o alinhamento ótimo de duas sequências pode ser encontrado em tempo de $O(n^2)$, onde n é o tamanho da maior sequência [CHLW05], apesar da comparação direta entre duas sequências de aminoácidos seja insuficiente para estabelecer um relacionamento genético entre duas proteínas [NW70].

Serão abordados a seguir três algoritmos exatos para encontrar o alinhamento ótimo entre pares de sequências.

3.1.1 Método de Needleman e Wunsch

O método de Needleman e Wunsch [NW70] encontra similaridade em sequências de aminoácidos de duas proteínas, possibilitando determinar as semelhanças entre as estruturas dessas proteínas. Geralmente essas sequências

são de proteínas com funções relacionadas e também são similares no aspecto visual, revelando coincidências.

Este método realiza o alinhamento global ótimo entre um par de sequências s e t, com $|s| = m$, $|t| = n$, em que m e n são os comprimentos das sequências s e t, respectivamente. É realizado o cálculo de uma matriz bidimensional *H*, onde o elemento $H_{i,j}$, representa a pontuação do melhor alinhamento entre os prefixos *s*[1 ... i] e *t*[1 ... j]. Para calcular esta matriz é utilizada programação dinâmica, sendo o início do processamento, a primeira linha e a primeira coluna são preenchidos com a penalidade de espaço gi, sendo que i é o tamanho da sequência não vazia terminando no $i-$ésimo elemento e g é a penalidade do espaço. As outras coordenadas são calculadas por meio da fórmula de recorrência descrita pela Equação 3.1.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + c_p(s_i, t_j) & \text{Custo para alinhar símbolos iguais ou diferentes} \\ H_{i-1,j} + g & \text{Símbolo } \textit{espaço} \text{ na sequência s} \\ H_{i,j-1} + g & \text{Símbolo } \textit{espaço} \text{ na sequência t} \end{cases} \tag{3.1}$$

Na Equação 3.1, $c_p(s_i, t_j)$ é o custo para alinhar o elemento s_i com t_j e g é o valor da comparação de qualquer elemento com o símbolo espaço.

A equação é aplicada para preencher a matriz *H*, do canto superior esquerdo para o canto inferior direito. Para se encontrar o alinhamento, mantém-se um ponteiro em cada célula da matriz indicando qual célula vizinha originou este valor, possibilitando assim percorrer estes ponteiros no sentido inverso, iniciando no canto inferior direito até chegar ao canto superior esquerdo, este passo é conhecido como *traceback*. Em alguns casos, o *traceback* poderá percorrer mais de um possível caminho, o que permite a geração de mais de um alinhamento global ótimo.

3.1.2 Algoritmo de Smith e Waterman

O algoritmo de Smith e Waterman [SW81] realiza o alinhamento local ótimo entre um par de sequências s e t, com $|s| = m$, $|t| = n$, em que m e n são os comprimentos das sequências s e t, respectivamente, muito parecido com o método de Needleman e Wunsch, porém com três diferenças. A primeira diferença está na inicialização da primeira coluna e da primeira linha, que são preenchidos com zeros, ou seja, $H(i, 0) = H(0, j) = 0$. A segunda diferença ocorre na equação de recorrência, em que se utiliza o valor zero para impedir que números negativos apareçam na matriz H. A ocorrência de um valor zero representa o começo de um novo alinhamento, pois se o alinhamento até determinado ponto for negativo, então é mais vantajoso começar um novo

alinhamento naquela posição. A nova equação de recorrência está descrita na Equação 3.2

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + c_p(s_i, t_j) & \text{Custo para alinhar símbolos iguais ou diferentes} \\ H_{i-1,j} + g & \text{Símbolo } \textit{espaço} \text{ na sequência s} \\ H_{i,j-1} + g & \text{Símbolo } \textit{espaço} \text{ na sequência t} \\ 0 \end{cases} \tag{3.2}$$

Na Equação 3.1, $c_p(s_i, t_j)$ é o custo para alinhar o elemento s_i com t_j e g é o valor da comparação de qualquer elemento com espaço.

A terceira mudança em relação ao Needleman e Wunsch, ocorre na forma de realizar o *traceback*. Em vez de iniciar na posição final $H_{m,n}$, o *traceback* inicia na posição $H_{i,j}$ onde ocorrer o maior valor na matriz H. Desta forma, obtém-se o melhor alinhamento desconsiderando prefixos e sufixos poucos significantes.

3.1.3 Método de Carrilo e Lipman

No contexto de alinhamentos usando soma de pares (SP), Carrilo e Lipman [CL88] projetaram um método para determinar um caminho γ=(S_1,...,S_n) entre as sequências S_1,...,S_n para $n > 2$, onde n é a quantidade de sequências, com uma redução significante de cálculos.

Seja $M(\gamma)$, a medida de similaridade entre as sequências S_1,...,S_n, que representa o *score* de cada caminho γ. Para cada $M(\gamma)$ existe no mínimo um caminho γ^*=(S_1,...,S_n), chamado de caminho ótimo.

Considere um reticulado N-dimensional $L = (S_1, \ldots, S_n)$. Cada vértice em L pode ter o canto final do sub-reticulado $L = (S_1(i_1), \ldots, S_n(i_n))$, onde para cada sequência S, $S(i)$ denota a sequência consistindo do primeiro i elementos de S, conforme é mostrado na Figura 3.1.

A ideia do método é encontrar recursivamente o caminho ótimo para todo sub-reticulado de $L = (S_1, \ldots, S_n)$. Inicialmente é computada a pontuação de cada possível caminho no cubo que tem o vértice no canto inicial. No próximo passo é calculada a pontuação mínima necessária para alcançar do canto inicial para os vértices adjacentes do cubo através de um caminho válido. Esse processo é repetido até ser calculada a pontuação mínima necessária para alcançar o canto final, isto é, a medida ótima do caminho $L = (S_1, \ldots, S_n)$. A pontuação ótima de cada vértice do reticulado é mantida em memória como um ponteiro para marcar cada passo da recursão. O menor segmento contribui para fazer o *traceback* através dos ponteiros para construir o caminho ótimo. É possível calcular γ^* para as sequências $(S_1, \ldots, S_n)$ de tamanho $(k_1, \ldots, k_n)$

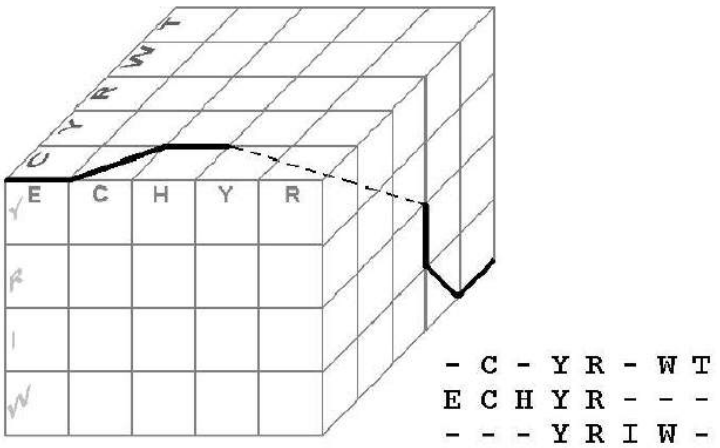


Figura 3.1: Um caminho em 3 dimensões, correspondente ao alinhamento de três sequências [KS06]

em $O\left(\sum_{i<j}^{n} k_i k_j\right)$ passos.

Normalmente o alinhamento de várias sequências implica realizar o alinhamento entre pares de $\binom{n}{2}$ das sequências. Ao observar γ^* é necessário considerar somente os caminhos γ em $L = (S_1, \ldots, S_n)$ que satisfaz a medida de similaridade entre as sequência S_i e S_j, chamado de caminho X_{kl}, para $l = (1, \ldots, n)$. Portanto o caminho γ no conjunto

$$X = \bigcap_{i<j}^{n} X_{ij}$$

são todos os possíveis caminhos candidatos para serem caminhos ótimos. Para considerar somente caminhos em X é necessário aplicar programação dinâmica somente em algumas sub-regiões Y de $L = (S_1, \ldots, S_n)$.

O conjunto Y_{ij}^{-1}, contém todos os caminhos em X_{ij} e o conjunto

$$Y = \bigcap_{i<j}^{n} Y_{ij}^{-1}$$

contém todos os caminhos em X. Geralmente Y é uma projeção p_{ij}, ou seja um subconjunto de Y_{ij}. Estas projeções são ilustradas na Figura 3.2. A região Y_{ij} pode ser determinada utilizando uma variação do algoritmo de programação dinâmica, quando o algoritmo é chamado novamente é calculado recursivamente a pontuação do caminho ótimo para cada sub-reticulado de L que compartilha o canto inicial em L. A recursão pode finalizar na direção oposta (do canto final para o canto inicial), calculando a pontuação do caminho ótimo para cada sub-reticulado que compartilha o canto final de L. Qualquer vértice em L, é conhecido o caminho ótimo do vértice L ao canto inicial, e o caminho ótimo do vértice L para o canto final. O caminho ótimo associado para cada vértice no reticulado é a informação necessária para decidir que ponto

em particular é um elemento de Y_{ij}.

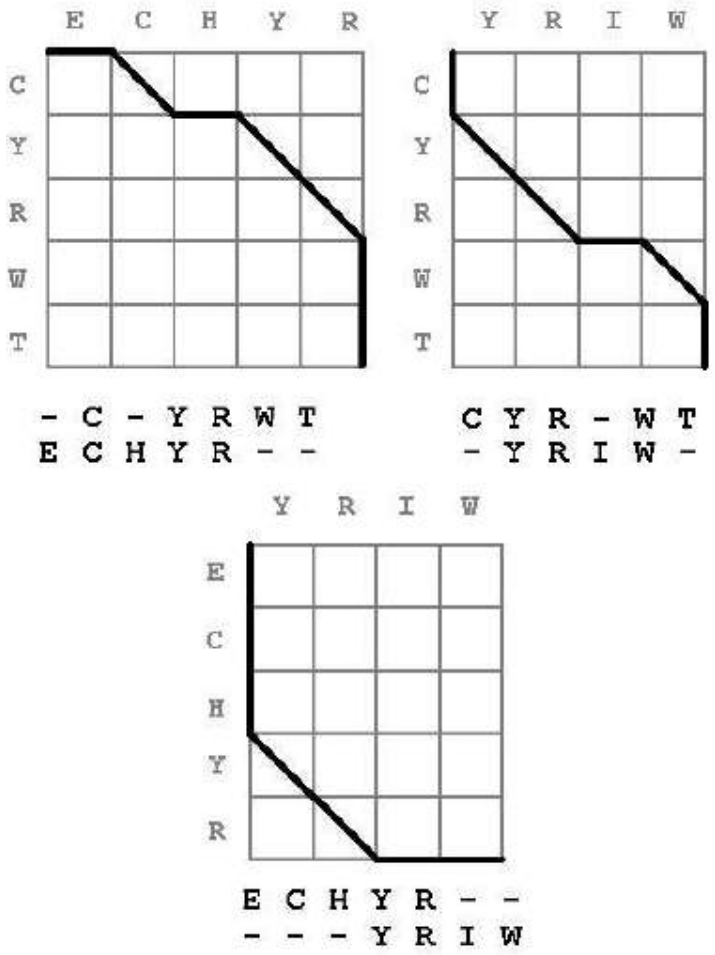


Figura 3.2: Projeção do caminho mostrado na Figura 3.1 [KS06]

Desta forma este método usa um limitante para restringir o tamanho da estrutura de programação dinâmica e encontra o alinhamento ótimo nesta área restrita. Mesmo com a redução de complexidade de tempo o algoritmo de Carrilo e Lipman ainda é exponencial [KS06].

O alinhamento ótimo A obtido pelo método de Carrillo e Lipman pode ser interpretado como um "refinamento" da solução A obtida por algum outro método de alinhamento de várias sequências [dB03].

3.2 Algoritmo de aproximação

Algoritmos de aproximação são algoritmos usados para encontrar soluções aproximadas em problemas de otimização, sendo geralmente associados a problemas NP-Difíceis. Esta classe de algoritmos ao contrário de procurar por uma solução ótima realiza a busca de uma solução considerada satisfatória, com certa garantia. Para calcular esta garantia, tem-se um conjunto $\mathbb{I}$ de instâncias de um problema de minimização, e OPT(I) um número associado a

cada instância $I \in \mathbb{I}$ e $\mathbb{A}(I) \geq$ OPT(I) um número computado por um algoritmo $\mathbb{A}$ com entrada I. Dize-se que $\mathbb{A}$ é uma α-aproximação para o problema se

$$\mathbb{A}(I) \leqslant \alpha.OPT(I),$$

para toda instância I, onde α é um *fator de aproximação* do algoritmo $\mathbb{A}$.

Na prática, muitas vezes é suficiente encontrar uma solução aproximada e em muitos casos isso pode ser feito muito rapidamente. Mesmo para problemas de otimização NP-Completos, podem existir algoritmos aproximados em tempo polinomial [Kan92].

3.2.1 Algoritmo JUNTA

Um método utilizado tanto em algoritmos de aproximação quanto em heurísticas para o problema de alinhar várias sequências é a junção de alinhamentos. A ideia básica deste método é "colar" dois ou mais alinhamentos para obter um alinhamento maior composto por todas as sequências dos alinhamentos menores [dB03].

Este método para junção de alinhamentos funciona tomando alinhamentos A_1, ..., A_r em que exatamente uma das sequências de entrada, por exemplo s, seja comum a cada par de alinhamentos e esteja na primeira linha de cada alinhamento, utilizando a sequência s como orientação, o método constrói um alinhamento A de todas as sequências. Esta sequência s comum em todos os alinhamentos é chamada de *sequência guia* ou sequência central.

O pseudocódigo deste algoritmo é apresentado no Algoritmo 1, sendo utilizado pelos métodos de alinhamento progressivo que incorporam ideias propostas por Feng e Doolittle [FD87].

Fixado i, seja $p_{i,j}$ a coluna do j-ésimo símbolo de s no alinhamento A_i para todo $j = 1, \ldots, n$, define-se $p_{i,0} = 0$ e $p_{i,n+1} = l_i + 1$. Seja $z_{i,j} = p_{i,j} - p_{i,j-1} - 1$, para todo $j = 1, \ldots, n+1$ e para todo $i = 1, \ldots, r$. A interpretação para $z_{i,j}$ de acordo com a definição é que $z_{i,j}$ é o numero de espaços que ocorre imediatamente antes de $s[j]$ no alinhamento A_i. Agora, define-se $z^*_j = max_i\{z_{i,j}\}$, para todo $j = 1, \ldots, n+1$, ou seja, o maior número de espaços que ocorre imediatamente antes de $s[j]$ em qualquer um dos alinhamentos A_i.

Um fato interessante no algoritmo JUNTA é que se dois caracteres consecutivos em alguma sequência de A_i estão separados por espaço, então eles continuam separados por espaço no alinhamento A'_i e, por consequência, também ficam separados no alinhamento A devolvido por JUNTA(A_1,...,A_r). Nas clássicas palavras de Feng e Doolittle, (*"once a gap, always a gap"*, uma vez

espaço, sempre espaço).

Algoritmo 1: Algoritmo JUNTA

Entrada: Alinhamentos $A_1, \ldots, A_r$ de comprimento $l_1, \ldots, l_r$ e com $k_1, \ldots, k_r$ sequências, respectivamente, em que exatamente uma sequência s é comum a cada par de alinhamentos; s é a primeira sequência de cada alinhamento.

Saída : Um alinhamento A entre as sequências de cada A_i, com A compatível com cada A_i, para $i = 1, \ldots, r$

```
1  for i ← 1 to r do
2      Determine p_{i,j};
3  for i ← 1 to r do
4      Determine z_{i,j};
5  for i ← 1 to n do
6      Determine z*_j;;
7  forall elementos de A_i do
8      Obtenha A'_i por inserção de colunas com espaços em A_i;
9  A ← 0. // A é um alinhamento vazio(sem linhas ou colunas).
10 forall elementos de A'_i do
11     Adicione todas as linhas de A' (com exceção da primeira) a A.
12 Adicione a primeira linha de A'_1 a A
13 return A
```

3.2.2 Algoritmo de Gusfield

O primeiro algoritmo de aproximação para o problema de alinhar várias sequências foi proposto por Gusfield [Gus93] e possui razão de aproximação α = 2-2/k. Para o alinhamento de *k* sequências, sendo *k*=2, tem-se α = 1, ou seja, o algoritmo de Gusfield também conhecido como algoritmo estrela, retorna o alinhamento ótimo entre duas sequências.

A ideia principal do algoritmo proposto por Gusfield é utilizar o alinhamento ótimo de pares de sequências para obter um alinhamento de várias sequências com qualidade aceitável. Para isso, dadas as sequências $s_1, s_2, \ldots, s_k$ o algoritmo opera fixando uma sequência s_c dentre as k sequências de entrada chamada *sequência central* e realiza o alinhamento ótimo de pares entre s_c e cada uma das k-1 sequências.

Resumidamente o algoritmo executa quatro passos sendo que no primeiro passo é realizado o alinhamento das sequências $s_1, s_2 \ldots, s_k$ dois-a-dois. No segundo passo é construída uma matriz de distância entre as sequências e escolhida a sequência centro s_c. No terceiro passo é realizado o alinhamento

ótimo dois-a-dois entre s_c e todas as outras. Por fim é construído um alinhamento de várias sequências começando com a sequência s_c e acrescentando uma nova sequência por vez, conforme pode ser visto no Algoritmo 2.

Fixando a sequência centro s_c, define-se $M(c) = \Sigma_{j \neq c} d(s_c, s_j)$, onde $d(s_c, s_j)$ é a distância entre s_c e s_j segundo alguma matriz de pontuação. O processo é repetido utilizando a sequência central e as demais sequências e para cada iteração o valor de M é calculado. O alinhamento com o menor valor de M dentre os k alinhamentos é devolvido como resposta. Para cada sequência k, B_i é o alinhamento ótimo entre a sequência centro s_c e a sequência s_i. Após o alinhamento estarem prontos é utilizado algoritmo JUNTA para juntar estes alinhamentos.

Algoritmo 2: Algoritmo ESTRELA

Entrada: $s_1, s_2, \ldots, s_k$ { sequências a alinhar }

Saída : Um alinhamento A das k sequências com $c(A) \leq (2 - 2/k)c(A^*)$, onde $c(A)$ é um alinhamento ótimo de $s_1, s_2, \ldots, s_k$ e $c(a)$ é o custo para realizar o alinhamento A

1 Calcule os alinhamentos entre todos os pares de sequências em { $s_1, s_2, \ldots, s_k$ }
2 **for** $i \leftarrow 1$ **to** k **do**
3 Calcule $M(i) = \Sigma_{j \neq i}\ d(s_i,\ s_j)$;
4 Seja $M = min_{i=1}^{k} \{M(i)\}$ e seja c tal que $M(c) = M$
5 **for** $i \leftarrow 1$ **to** k **do**
6 Seja B_i o alinhamento ótimo entre s_c e s_i
7 **return** *JUNTA*($B_i, \ldots, B_k$), $\forall\ 1 \leq i \neq j \leq k$

No algoritmo de Gusfield, supondo que as k sequências possuam comprimento l o primeiro passo leva tempo O(k^2l^2). O cálculo de cada $M[i]$ leva tempo O(k) e como $M[i]$ é calculado para cada valor de i, o segundo passo pode ser completado em O(k^2). A denominação de m no passo 3 pode ser feita em O(k). A junção no passo 4 é calculada em tempo O(k^2l). Portanto, para k sequências $s_1, s_2 \ldots, s_k$ de tamanho l cada, o algoritmo de Gusfield devolve um alinhamento em tempo O(k^2l^2).

CAPÍTULO

4

Modelos de paralelismo

Tradicionalmente, os softwares são escritos para computação em série. Um problema é dividido em uma série de instruções, sendo estas instruções executadas sequencialmente um após outra em um único processador. Apenas uma instrução pode executar em um determinado momento.

De maneira mais simples, na computação paralela são utilizados simultaneamente vários recursos de computação. O problema é quebrado em partes e podem ser resolvidos simultaneamente em diferentes processadores, empregando um mecanismo geral de controle ou coordenação.

A memória principal de um computador paralelo é compartilhada ou distribuída. Ao utilizar memória compartilhada, vários processadores acessam um único espaço de endereçamento. A GPU (do inglês, *Graphics Processing Unit*, Unidade de Processamento Gráfico) aproveita-se desta arquitetura de memória [Coo12]. Ao utilizar memória fisicamente distribuída entre os processadores, cada memória local só pode ser acessada pelo seu processador e a sincronização requer comunicação entre processadores através de troca de mensagens. O MPI (do inglês, *Message Passing Interface*) é baseado principalmente nesta arquitetura [GLS99]. Existem ainda arquiteturas de memória híbridas que utilizam memória compartilhada e distribuída.

O tamanho do processo que está sendo executado é determinado pelo programador e pode ser caracterizado através de sua granulosidade, isto é, a razão entre tempo de computação e o tempo de comunicação. Na granulosidade grossa, cada processo gasta relativamente uma grande quantidade de tempo executando instruções sequenciais. Quando o processo executa poucas instruções sequenciais entre as operações de comunicação temos a granulosidade fina.

Neste capítulo serão discutidos brevemente sobre os modelos de paralelismo em GPU e MPI, que foram utilizados para realizar o alinhamento de várias sequências.

4.1 GPUs

Considerando o supercomputador mais rápido atualmente, o *Thianhe-2* possui uma performance de 33,86 *petaflops/s* (quadrilhões de cálculos por segundo), composto por milhares de GPUs, sendo listado no site top 500 (http://www.top500.org). Ambos os supercomputadores e computadores *desktop* estão seguindo a computação heterogênea utilizando CPU (Unidade de Processamento Central) e GPU [Coo12].

As GPUs são dispositivos presentes em PCs mais modernos. Elas proporcionam um número de operações gráficas para a CPU, como renderizar uma imagem na memória e em seguida exibir a imagem na tela.

Em 2007, a NVIDIA viu uma oportunidade de trazer GPUs para o destaque, acrescentando uma ferramenta de programação que é denominado CUDA (do inglês, *Compute Unified Device Architecture*). Um programa em CUDA possui a capacidade de executar instruções de programação geral na GPU. Ele pode ser dividido em duas partes: o código *host* que é executado na CPU e o código *device*, que é executado na placa de vídeo. Os *Kernels* são a forma pela qual o código da CPU é capaz de chamar o código na GPU, sendo possível assim a interação entre eles. Todas as instruções executadas na placa gráfica é escrito dentro de um *kernel*.

Um dos problemas com processadores modernos de hoje é que eles atingiram um limite de velocidade de *clock* em cerca de 4 GHz [Coo12]. Neste ponto, eles geram calor demais para a tecnologia atual e requerem soluções de refrigeração caros. Isso ocorre porque à medida que se aumenta a velocidade do *clock*, o consumo de energia também aumenta, com isso os dois principais fabricantes de processadores de PC, a Intel e AMD tiveram que adotar uma abordagem diferente. Eles têm sido forçados a aumentar a adição de mais núcleos para processadores, em vez de tentar continuamente aumentar o *clock* da CPU e/ou extrair mais instruções por *clock* através de paralelismo no nível de instruções.

A partir de 2009 começamos a ver a diferença entre o poder computacional de CPU e GPU, para aplicações capazes de explorar este paralelismo, quando a GPU quebra a barreira de teraflop conforme mostra Figura 4.1.

Nesse momento, passou-se do *hardware* G80 para o *hardware* G200 e em seguida, em 2010 evoluiu-se para o *hardware* Fermi, sendo conduzido pela introdução de *hardware* massivamente paralelo. O *hardware* G80 é um dis-

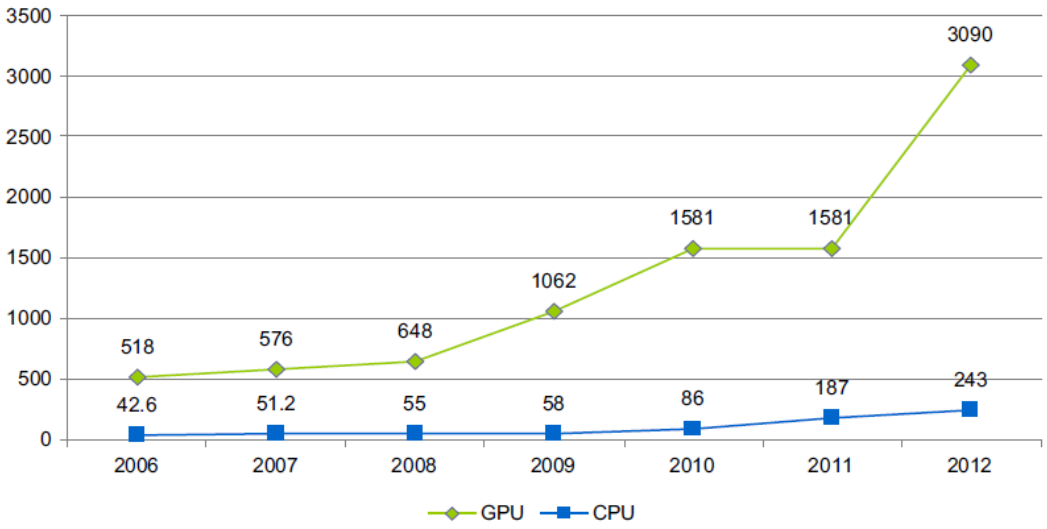


Figura 4.1: Pico de performance em *gigaflops* entre CPU e GPU [Coo12]

positivo CUDA com 128 núcleo e o *hardware* G200 contém 256 núcleos, já o Fermi é um dispositivo CUDA com 512 núcleos.

É possível observar na arquitetura típica de um *Core 2*, conforme apresentado na Figura 4.2, que todos os dispositivos da GPU são ligados ao processador através de um barramento PCI-E (do inglês, *Peripheral Component Interconnect Express*), tendo assim um barramento PCI-E 2.0, com uma taxa de transferência de 5GB/s.

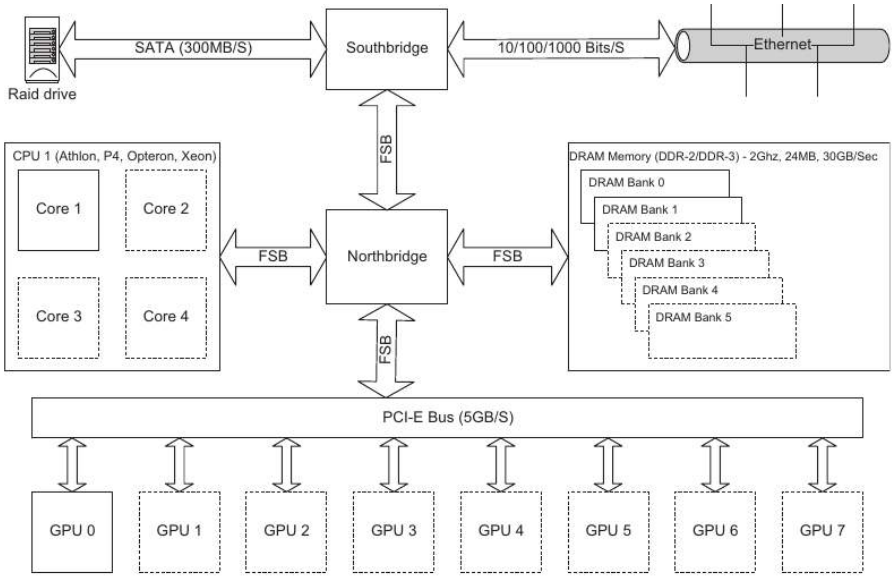


Figura 4.2: Layout típico de uma série do *Core 2* [Coo12]

No entanto, para obter dados do processador, é preciso passar pelo dispositivo *Northbridge* sobre o barramento FSB (do inglês, *Front-Side Bus*). O FSB pode executar qualquer coisa num *clock* de até 1600 MHz, embora possa ser muito lenta em alguns casos. A memória também é acessada através do *Northbridge* e os periféricos através do chipset *Northbridge e Southbridge*. Os

Northbridge lida com todos os componentes de alta velocidade como memória, CPU, Ligações PCI-E, etc. Os chips *Southbridge* lidam com os dispositivos mais lentos como discos rígidos, USB, teclado, conexões de rede, etc. Claro, é perfeitamente possível conectar um controlador de disco rígido em uma conexão PCI-E e na prática, esta é a única maneira de conseguir RAID de alta velocidade de acesso a dados. O PCI-E é um barramento interessante como, ao contrário de seu predecessor, PCI (do inglês, *Peripheral Component Interconnect*), é baseado na largura de banda. No velho sistema PCI, cada componente pode usar a largura de banda total do barramento, mas apenas um dispositivo por vez. Assim, quanto mais cartões forem adicionados, cada cartão iria receber a menor largura de banda. Uma topologia que resolveu este problema foi a criação de faixas PCI-E. Estas faixas são ligações de alta velocidade em série, que podem ser combinadas para formar X1, X2, X4, X8 ou X16 ligações. Com esta configuração, tem-se um barramento full-duplex de 5GB/s, o que significa a mesma velocidade de *upload* e *download*, ao mesmo tempo. Assim, é possível transferir 5GB/s para o cartão, ao mesmo tempo em que recebemos 5GB/s a partir de uma placa. No entanto, isto não significa que é possível transferir 10GB/s para o cartão se não estamos recebendo todos os dados (ou seja, a largura de banda não é cumulativa).

Em um ambiente típico de supercomputador, ou até mesmo em aplicações *desktop*, é manipulado um grande conjunto de dados. Um supercomputador pode lidar com petabytes de dados. Um computador *desktop* pode lidar com pouco ou vários GB de vídeo de alta definição. Em ambos os casos, não há dados consideráveis para buscar em periféricos conectados. Um simples disco rígido de 100 MB/s irá carregar 6 GB de dados em um minuto. Neste ritmo, serão necessárias duas horas e meia para ler todo o conteúdo de um disco de 1 TB. Se estiver usando MPI (*Message Passing Interface*), comumente usado em *clusters*, a latência para este arranjo pode ser considerável, caso as ligações Ethernet estejam conectadas ao *Southbridge* em vez de estarem ligadas ao barramento PCI-E.

Quando não havia interface direta entre GPU e MPI, todas as comunicações em um sistema desse tipo eram encaminhadas através do barramento PCI-E para a CPU e vice-versa. A tecnologia *GPU-Direct*, disponível no CUDA 4.0 SDK, resolveu este problema e agora é possível que determinados cartões *InfiniBand* falem diretamente com a GPU sem passar pela CPU primeiro. Esta atualização para o SDK também permite comunicação direta com a GPU.

Conforme é mostrado na Figura 4.3, o hardware da GPU consiste de uma série de blocos principais, como memória, classificadas em memória global, memória constante e memória compartilhada, *Streaming Multiprocessor* (SMs) e *Stream Processor* (SPs). É possível observar que a GPU é constituída

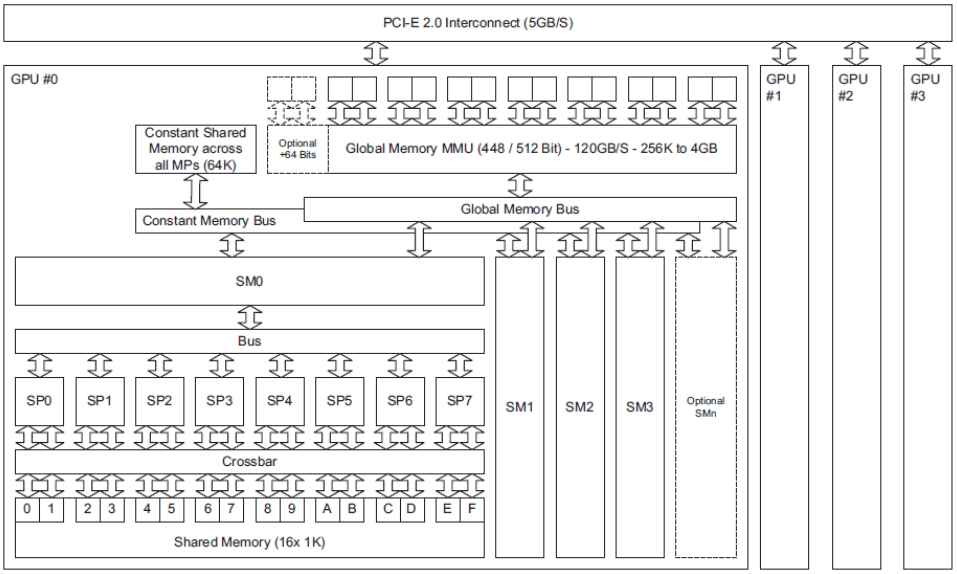


Figura 4.3: Diagrama de bloco de um cartão GPU (G80/GT200) [Coo12]

por uma série de SMs, cada um com N núcleos. Este é o aspecto chave que permite dimensionamento de processador. Um dispositivo GPU consiste em um ou mais SMs. Adicione mais um par de SMs e a GPU será capaz de processar mais tarefas.

4.2 MPI

Um padrão para comunicação de dados em computação paralela muito utilizado é o MPI [GLS99]. Existem várias modalidades de computação paralela, e dependendo do problema que se está tentando resolver, pode ser necessário passar informações entre os vários processadores de um *cluster*, e o MPI oferece uma infraestrutura para essa tarefa.

No padrão MPI, uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos. Inicialmente, na maioria das implementações, um conjunto fixo de processos é criado. Porém, esses processos podem executar diferentes programas. Por isso, o padrão MPI é algumas vezes caracterizado como MPMD (*Multiple Program Multiple Data*). Elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. Geralmente em um ambiente de computação paralela o número de processos é fixo. Os processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro). Além disso, um grupo de processos pode invocar operações coletivas de comunicação para executar operações globais. O MPI é capaz de realizar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*) que permitem ao usuário

definir módulos que encapsulem estruturas de comunicação interna.

Pode-se citar como vantagens de utilização deste modelo de computação paralela a portabilidade, ótimo desempenho e a possibilidade de explorar paralelismos em diversos problemas computacionais.

CAPÍTULO

5

Implementações paralelas para alinhamento de várias sequências

Muitas heurísticas são propostas para realizar o alinhamento de várias sequências em arquiteturas paralelas, incluindo contribuições em *cluster* e GPU. Por outro lado, existem poucos algoritmos exatos para o alinhamento ótimo de várias sequências em plataformas de alto desempenho.

Neste capítulo serão discutidos algumas implementações exatas e heurísticas para o alinhamento de várias sequências, que exploram o paralelismo utilizando GPU e MPI.

5.1 Algoritmo exato

Helal [HEGMG08] apresenta uma forma de distribuir o cálculo do algoritmo de programação dinâmica para realizar o alinhamento de várias sequências utilizando *Peer-to-Peer* (P2P) e MPI. A implementação proposta por Helal, conseguiu pontuações melhores que implementações heurísticas como Clustal W [THG94], T-COFFE [NHH00] dentre outras, alcançando *speedup* 5 vezes melhor que implementações utilizando o método Mestre/Escravo. Os teste foram realizados utilizando 5 sequências com tamanho máximo de 41 símbolos.

5.2 Heurísticas

Um algoritmo heurístico é um algoritmo que funciona bem em casos práticos, mas não existe uma garantia de que a solução devolvida esteja próxima

da solução ótima. Normalmente, heurísticas são fundamentadas em métodos não rigorosos, mesmo que forneçam resultados satisfatórios.

Esta seção apresenta as implementações heurísticas paralelas para o alinhamento de várias sequências. As implementações utilizam técnicas paralelas em CUDA e MPI.

5.2.1 MSA-CUDA

O MSA (do inglês, *Multiple Sequences Alignment*, Alinhamento de Várias Sequências), tipicamente consiste em três estágios conforme mostrado na Figura 5.1. O primeiro estágio calcula a pontuação para cada par de sequências usando programação dinâmica. No segundo estágio é calculada a árvore evolutiva da matriz de distância usando o método de reconstrução filogenética. Esta árvore é usada como uma árvore guia e no terceiro estágio é realizado o alinhamento progressivo dos pares de sequências para o MSA final [LSM09a].

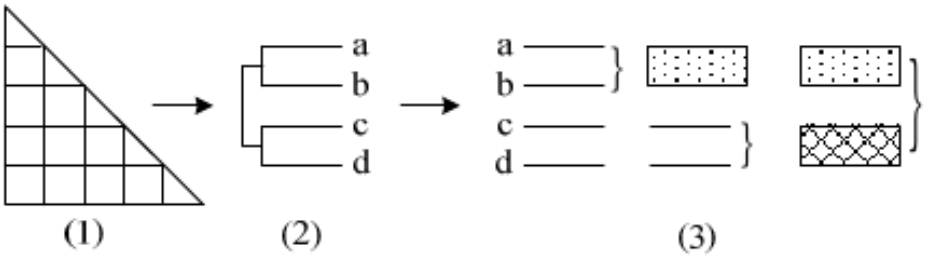


Figura 5.1: Três estágios do alinhamento progressivo. (1) matriz de distância, (2) árvore guia, (3) alinhamento progressivo [LSM09a]

Liu realizou a implementação do MSA usando arquitetura CUDA e o algoritmo de Myers [MM88] que encontra o "ponto do meio ótimo" de um alinhamento ótimo. A implementação sequencial do algoritmo usa métodos recursivos de divisão e conquista, mas como o CUDA não permite recursão, foi desenvolvida uma nova implementação iterativa baseada em pilha. O MSA-CUDA usa esta implementação para ambos os alinhamentos de pares no primeiro estágio e no alinhamento de perfil-perfil no terceiro estágio. No primeiro estágio foram utilizados duas abordagens, paralelização inter-tarefa e paralelização intra-tarefa. Na paralelização inter-tarefa cada tarefa é atribuída para exatamente uma *thread* e um bloco de tarefas é executado paralelamente por diferentes *threads* dentro de um bloco de *threads*. Já paralelização intra-tarefa cada tarefa é atribuída para um bloco inteiro de *threads* que ajudam na performance da tarefa paralela.

Na paralelização do segundo estágio, a construção da árvore guia, foi dividida em dois estágios. O estágio 2a realiza a reconstrução da árvore não enraizada e o estágio 2b calcula o enraizamento da árvore e dos pesos das

sequência. A implementação em CUDA utiliza um vetor que armazena a relação entre os nós da direita, da esquerda e de seu pai. Um bloco de *threads* é designado para calcular a diferença dos comprimentos dos ramos da esquerda e da direita de um nó. Cada *thread* no bloco de *threads* é atribuído para calcular em separado o subconjunto de nós folhas. Para realizar estes cálculos com a estrutura da árvore, a memória compartilhada é utili zada para armazenar o resultado de todas as *threads* em um bloco de *threads*.

O terceiro estágio é paralelizado utilizando o método de alinhamento progressivo, iniciando das folhas até a raiz de uma árvore enraizada. Cada nó da árvore corresponde a um perfil do alinhamento, produzido a partir das sequências alinhadas na subárvore da esquerda e da direita. Três vetores auxiliares são utilizados para registrar os índices de seus filhos e uma *flag* que indica se o respectivo alinhamento foi executado.

5.2.2 Clustal e suas variantes

Clustal [HS88] é um algoritmo sequencial muito popular, utilizado desde 1988 para o alinhamento de várias sequências, baseado no método de alinhamento progressivo. O Clustal originalmente executava em computadores IBM, sofrendo diversas mudanças ao longo dos anos, buscando melhoria no alinhamento de várias sequências. O Clustal W [THG94] é base para todas as demais versões que surgem do Clustal após 1994 [LBB+07].

Assim como apresentado na subseção 5.2.1, o Clustal W é executado em três estágios. Na versão inicial do Clustal, o primeiro estágio é calculado usando o método de aproximação rápida, permitindo que um grande número de sequências seja alinhado até mesmo em microcomputadores. No Clustal W é possível escolher entre este método de aproximação rápida e o método de programação dinâmica mais lenta, porém com pontuações mais precisas.

No segundo estágio, as árvores utilizadas para orientar o processo de alinhamento final das várias sequências são calculadas a partir da matriz de distâncias calculados no primeiro estágio, fazendo uso do método *Neighbour-Joining* [SN87].

O procedimento básico do terceiro estágio consiste em utilizar uma série de pares de alinhamentos, para alinhar grupos cada vez maiores de sequências, seguindo a ordem de ramificação na árvore guia, avançando a partir das pontas da árvore enraizada para a raiz. Uma das melhorias desenvolvidas no Clustal W é aplicado nesta fase final do alinhamento progressivo. Inicialmente os símbolos espaços são calculados dependendo do peso na matriz de distância, levando em consideração a similaridade e o tamanho da sequência. Em seguida é realizada a derivação de símbolos espaços sensíveis a abertura de penalidade em cada posição, em cada grupo de sequências já alinhadas

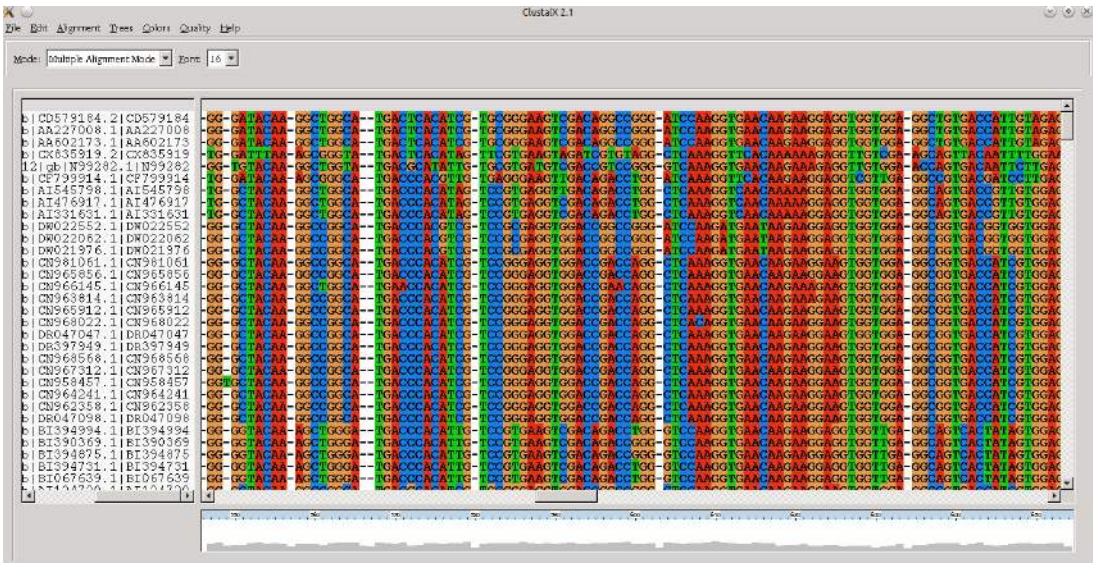


Figura 5.2: *Screenshot* do programa ClustalX 2.1 no linux.

que variam a medida que novas sequências são adicionadas. A última modificação permite atrasar a adição de sequências muito divergentes, até o final do processo de alinhamento, quando todas as sequências mais estreitamente relacionadas já foram alinhadas. A complexidade total do Clustal W é apresentado na Tabela 5.1, sendo $O(n^2 l_{ave}^2 + n^3)$, onde n é a quantidade de sequências a serem alinhadas e l_{ave} é o tamanho médio de todas as sequências.

Estágio	O(*time*)
Matriz de Distâncias	$O(n^2 l_{ave}^2)$
Árvore Guia	$O(n^3)$
Alinhamento Progressivo (cada iteração)	$O(nl_{ave} + l_{ave}^2)$
Alinhamento Progressivo (total)	$O(nl_{ave}^2 + n^2 l_{ave})$
Total	$O(n^2 l_{ave}^2 + n^3)$

Tabela 5.1: Complexidade de execução do Clustal W nos três diferentes estágios [LSM09b].

Após o desenvolvimento do Clustal W, em 1997 Thompson [TGP+97] desenvolve o Clustal X conforme é mostrado na Figura 5.2, que é uma versão gráfica do Clustal W, encontrando-se atualmente na versão 2.1. O Clustal X tem as mesmas funcionalidades do Clustal W, sendo capaz de alinhar conjuntos de dados de tamanho médio muito rapidamente. Duas novas funcionalidades têm sido incluídas no Clustal W e Clustal X 2.0, permitindo realizar alinhamentos mais rápidos em quantidade maiores de sequências e com maior precisão no alinhamento final. O uso do método UPGMA na segunda fase do alinhamento, permite realizar o agrupamento de 10.000 sequências em um computador *desktop* padrão em menos de um minuto, enquanto utilizando o método *Neighbour-Joining* leva mais de uma hora [LBB+07]. A segunda funcionalidade adicionada ao Clustal é um refinamento na pontuação

da soma de pares ponderada (*Weighted Sum of Pairs* - WSP), onde em cada iteração do terceiro estágio, cada sequência é removida do alinhamento e realinhado. Se a pontuação do WSP for reduzida, então o alinhamento resultante é mantido, permitindo assim mais precisão no alinhamento final, porém consumindo mais tempo.

Em 2003, surge uma nova versão do Clustal W, explorando a paralelização em *clusters* e utilizando a biblioteca MPI, chamado de ClustalW-MPI [Li03]. A paralelização da matriz de distância no primeiro estágio é feita utilizando a estratégia *fixed-size chunking* [Hag97], onde lotes fixos de tarefas são alocados a estações de trabalho ociosas. Com lotes grandes é minimizada a sobrecarga de comunicação entre as estações, porém pode ocorrer um tempo ocioso no processamento, enquanto a utilização de lotes menores pode reduzir o tempo ocioso, porém com uma sobrecarga na comunicação.

A árvore guia do segundo estágio do ClustalW-MPI é executado em $O(n^2)$, seguindo o método *Neighbour-Joining* [SN87] com poucas alterações em relação ao Clustal W, ou seja, este estágio não foi paralelizado.

Por fim, o último estágio de alinhamento progressivo é realizado com uma mistura entre a granulosidade grossa e granulosidade fina. Na granulosidade grossa, todos os nós externos na árvore guia são alinhados em paralelo. Uma árvore balanceada permite velocidade estimada de $N/logN$, onde N é o número de nós da árvore.

5.2.3 T-COFFE

T-Coffe [NHH00], (do inglês, *Tree-based Consistency Objective Function for alignment Evaluation*, função objetivo consistente baseado em árvore para avaliação de alinhamento), tem duas características principais. Primeiro ele fornece uma maneira simples de gerar alinhamento de várias sequências, usando fonte de dados heterogênea. Os dados destas fontes são fornecidos ao T-Coffe através de uma biblioteca de alinhamento global e local em pares, conforme ilustrado na Figura 5.3.

A segunda característica do T-Coffe é o método de otimização, usado para encontrar os melhores ajustes no alinhamento de várias sequências, usando a estratégia de alinhamento progressivo, similar ao utilizado pelo Clustal W [THG94]. Ao contrário do Clustal W o T-Coffe tem a capacidade de considerar as informações de todas as sequências durante cada etapa do alinhamento, não apenas aquelas que estão sendo alinhadas nessa fase.

A biblioteca primária contém o conjunto de todos os alinhamentos em pares das sequências a serem alinhadas. Logo, existem nesta biblioteca $n(n\text{-}1)/2$ elementos.

Os alinhamentos globais são construídos com o Clustal W com a finalidade

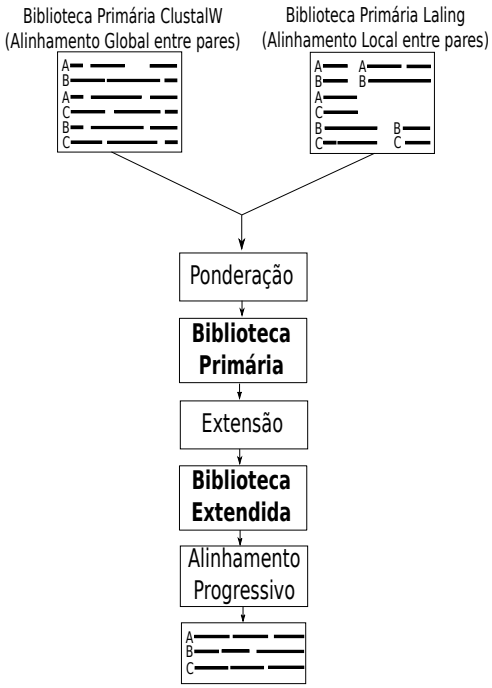


Figura 5.3: Layout do método T-Coffe [NHH00]

de obter o tamanho do maior alinhamento. O alinhamento local é construído utilizando a ferramenta *Laling* utilizando as 10 sequências com maior pontuação que não fazem interseção no alinhamento local. Ambas as ferramentas são executadas com seus parâmetros padrões. Esta biblioteca realiza o alinhamento de várias sequências utilizando somente os pares de sequências mais confiáveis, isto é, aqueles com as maiores pontuações.

O objetivo é combinar as informações do alinhamento local e global realizado pelas duas ferramentas, num simples processo de adição. Se existir qualquer duplicidade de alinhamento entre as duas ferramentas as sequências serão fundidas em uma única entrada com peso igual a soma dos dois pesos.

5.2.4 Algoritmo de streaming

O algoritmo de *streaming* é um algoritmo proposto por Liu [LSVMW07], baseado em programação dinâmica, usado para calcular o alinhamento ótimo entre várias sequências biológicas utilizando CUDA. O tempo gasto para calcular grandes conjuntos de dados varia de acordo com o tamanho da sequência biológica a ser computada. Os algoritmos de alinhamentos estão sendo redesenhados para aproveitar as características das arquiteturas paralelas para obter redução do tempo de execução.

O programa Clustal W, que usa alinhamento progressivo, revelou que mais de 93% do tempo no processamento é gasto no primeiro estágio (cálculo da matriz de distância). Portanto, Liu concentrou-se no desenvolvimento de um

algoritmo de fluxo para este estágio, seguindo a definição de distância entre duas sequências, utilizado pelo programa Clustal W.

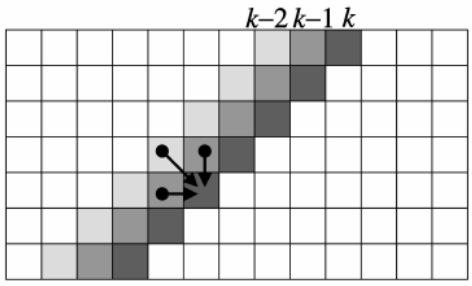


Figura 5.4: Relação de dependência entre os dados na matriz de programação dinâmica de Smith-Waterman [LSVMW07]

Lembre-se que o algoritmo de programação dinâmica de Smith-Waterman [SW81] encontra a subsequência mais similar de duas sequências (alinhamento local). Este algoritmo compara duas sequências para calcular a distância que representa o mínimo custo para transformar um segmento em outro segmento. Uma abordagem explorada em programação dinâmica paralela é o fato que todos os elementos de sua diagonal na matriz de Smith-Waterman podem ser calculados independentemente e em paralelo conforme apresentado na Figura 5.4.

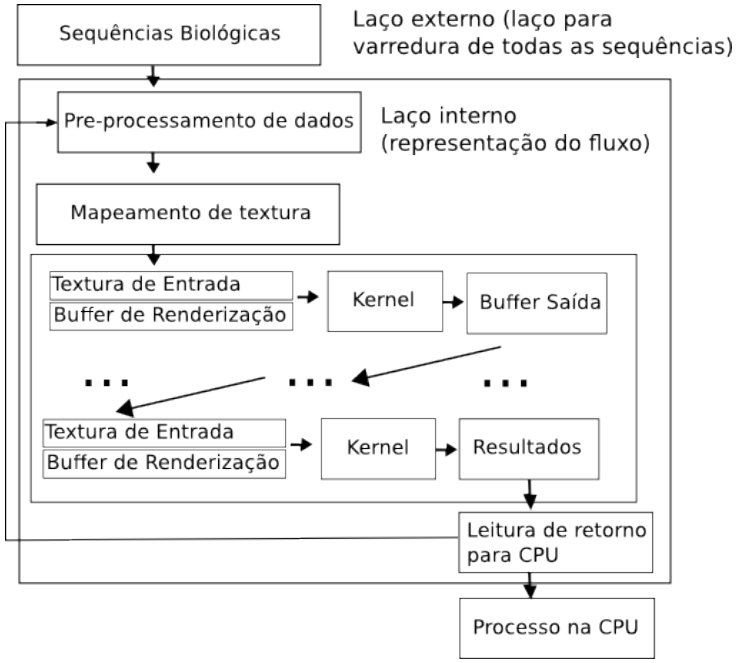


Figura 5.5: Fluxo do algoritmo para alinhamento baseado em GPU [LSVMW07]

A ideia básica do método é calcular a matriz de programação dinâmica usando apenas a diagonal. Mostra-se na Figura 5.5 o fluxo de execução do algoritmo para busca de sequências, consistindo em dois laços. O laço externo realiza a leitura de novas sequências dentro da memória de textura que serão

alinhadas. O laço interno calcula a matriz da programação dinâmica correspondente na diagonal. A diagonal é armazenada na memória de textura. Os *kernels* são usados para implementar as operações aritméticas especificadas pela relação de recorrência. Para alinhar duas sequências de tamanho l_1 e l_2 na GPU, em uma fase de pré-processamento as sequências e a matriz de substituição são carregadas na memória de textura. A matriz de programação dinâmica é calculada em l_1 + l_2 -1 *kernels* e a nova diagonal calculada é armazenada na memória de textura. Como a diagonal k depende da diagonal $k-1$ e $k-2$, as três diagonais são armazenadas em *buffers* separados. O *kernel* subsequente realiza a leitura de duas diagonais já calculadas anteriormente na memória e realiza outro ciclo de processamento, realizando várias vezes este processo até terminar o cálculo da matriz de programação dinâmica.

Este método tem como propósito acelerar a comparação de sequências, por exemplo, é possível encontrar com maior celeridade em um banco de dados de proteínas, sequências que possuem funcionalidades conhecidas. Todos os alinhamentos são classificados pela pontuação máxima na matriz de similaridade. Com isto o *traceback* somente é executado na GPU com as sequências de maior pontuação.

O autor compara o tempo de execução do Clustal W em um Pentium 4 de 3GHz CPU com sua implementação proposta em um Pentium 4 com 3GHz e GPU Nvidia GeForce GTX 512 para uma quantidade variando de 400 a 1.000 sequências de proteínas, foi obtido um *speedup* em média de 12 vezes comparado com o primeiro estágio Clustal W.

5.2.5 Paralelização do método centro estrela

Algumas abordagens de alinhamento de várias sequências utilizam algoritmos tais como: alinhamento por soma de pares (SP) [SPD97], alinhamento por consenso [Wat86] e alinhamento por árvore [AL89]. Apesar de serem paralelizados com alguma garantia de aproximação, estes algoritmos gastam muito tempo executando o método centro estrela proposto por Gusfield [Gus93], para encontrar a sequência centro. Dado um conjunto $S = \{S_1, S_2, \ldots, S_k\}$ de k sequências, a sequência centro $S_c \in S$ é uma sequência em S que minimiza $\sum_{j=1}^{k}$ D(S_c, S_j), onde D(S_c, S_j) denota a menor distância de edição entre as sequências S_c e S_j [Gus97].

Sahoo [SKP09], sugerem um algoritmo paralelo de granulosidade grossa para o método centro estrela a ser utilizado no alinhamento de várias sequências. A arquitetura proposta é baseada na abordagem *mestre-escravo*, sendo que o módulo central é executado no computador principal, responsável por realizar o controle de execução, fluxo de dados para os escravos e dos escravos e a decisão da sequência centro. A troca de mensagens entre o computador

principal e os escravos é realizada utilizando a biblioteca MPI, onde todas as transmissões entre o computador principal e os escravos acontecem no modo não bloqueante.

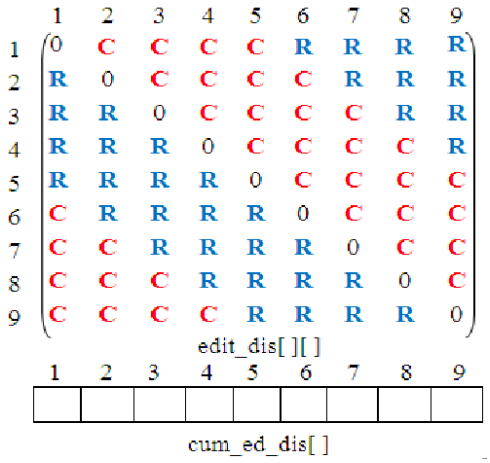


Figura 5.6: Matriz de todas as distâncias de edição e soma de distâncias de edição [SKP09]

Algoritmo 3: Algoritmo Paralelo Centro Estrela

Processo principal P_0 distribui $\lfloor no\_seq/2 \rfloor + 1$ lotes de pares de sequências para todos os outros processos escravos P_i; $1 \leq i \leq (\lfloor no\_seq/2 \rfloor - 1)$;

/* Cálculo da distância de edição para ambos processos *master* e *slave* */

$i \leftarrow process\_id$;

for $k \leftarrow 0$ **to** $(\lfloor no\_seq/2 \rfloor - 1)$ **do**

  $j \leftarrow (i + k + 1)$ % no_seq;

  $ed\_dis[i][j] \leftarrow edit\_distance(S_i, S_j)$; /* Armazena a distância de edição dos pares usando programação dinâmica entre as sequências S_i e S_j */

  Cada processo P_i enviará o cálculo da distância de edição para o processo P_j usando protocolo não bloqueante *send* & *receive*

/* Calcula a soma da distância de edição por cada processo na linha/coluna */

for $k \leftarrow 0$ **to** $(\lfloor no\_seq/2 \rfloor - 1)$ **do**

  $cum\_ed\_dis+ = ed\_dis[i][k]$;

/* Encontra a sequência centro pelo *master* */

$min \leftarrow cum\_ed\_dis[0]; c\_star = 0$;

for $i \leftarrow 1$ **to** $(no\_seq/2 - 1)$ **do**

  $min \leftarrow cum\_ed\_dis[i]$; $c\_star \leftarrow i$;

Processo *Master* declara c_star como sequência centro;

A pontuação ótima do alinhamento global entre pares é calculada utilizando programação dinâmica. Este passo é repetido iterativamente para todos $n(n\text{-}1)/2$ pares de sequências, calculando a pontuação da distância de

edição para as n sequências a serem alinhadas. Todas estas pontuações são armazenadas em um vetor de 2 dimensões (ed_dist) e a soma de todas as distâncias de edição para as linhas/colunas são armazenados em um vetor de uma dimensão (cum_ed_dist) conforme é mostrado na Figura 5.6. O processo responsável em calcular as distâncias de edição nas linhas da matriz (ed_dist), tem suas entrada marcada com "C" e o restante são calculados e recebidos de outros processos e são marcados com "R" como é mostrado na Figura 5.6. O menor valor presente no vetor (cum_ed_dist) é declarado a sequência centro (S_c) para o método centro estrela.

O algoritmo paralelo de granulosidade grossa proposto por Sahoo, descrito pelo Algoritmo 3 foi implementado utilizando MPI. O sistema de paralelização é composto de um conjunto de *n* multiprocessadores simétricos: p_1, p_2, ..., p_{n-1}.

A complexidade do método centro estrela é O(k^2l^2). Na paralelização de Sahoo cada processador p alinha um par de sequência tendo complexidade de $O(k^2l^2/p)$. A comunicação entre os processadores gasta $O(l)$ operações, consequentemente o algoritmo é de custo ótimo.

O experimento de Sahoo visou descobrir como o número de sequências afeta o desempenho do sistema em função do número de *hosts* disponíveis. Neste experimento a quantidade de sequências alinhadas varia de 7 a 16 e o comprimento da sequência foi fixado em 8Kbps e 1Kbps. O *speedup* para 4 processadores foi quase linear chegando a 90%, incentivando a implementação paralela do método centro estrela para o alinhamento de várias sequências.

5.2.6 Alinhamento em Clusters de GPU

Recentemente Truong [TLS+14], apresentaram quatro implementações em GPU para o alinhamento de várias sequências usando o algoritmo de Needleman e Wunsch em larga escala. Cada estratégia de implementação explora o modo de acesso a memória e a utilização do *hardware* da GPU. Três implementações (*TiledDScan-mNW*, *DScan-mNW* e *RScan-mNW*) são de propósito geral e a quarta implementação (*LazyRScan-mNW*) é otimizada para problemas que necessitam de boa performance na realização das operações de *traceback*.

O método *TiledDScan-mNW* realiza o alinhamento das várias sequências operando nas diagonais, realizando várias invocações de *kernel*. No entanto para cada chamada de *kernel*, várias matrizes de alinhamento são simultaneamente calculadas utilizando blocos de *threads*. Conforme mostra a Figura 5.7 (a), tem-se três alinhamentos simultâneos: (seq_1, seq_2), (seq_1, seq_3) e (seq_1, seq_4). Na primeira iteração as três fatias do topo esquerdo são processados em paralelo por três blocos de *threads* e portanto mapeado dentro de três *streaming*

multiprocessor (SM_1, SM_2, SM_3). Na segunda iteração, as fatias da segunda diagonal menor são processadas em paralelo por seis blocos de *threads* e mapeados dentro dos *streaming multiprocessor* ($SM_1 - SM_6$).

A segunda implementação *DScan-mNW* opera na diagonal com uma simples chamada de *kernel* conforme mostra a Figura 5.7 (b), neste caso cada alinhamento da matriz é atribuído a um bloco de *threads* e mapeado dentro da SM que realiza o cálculo de cada diagonal mapeado dentro do núcleo. O padrão de cálculo do *DScan-mNW* é similar a implementação paralela intra-tarefa do algoritmo de Smith-Waterman proposto por Liu [LSM09a].

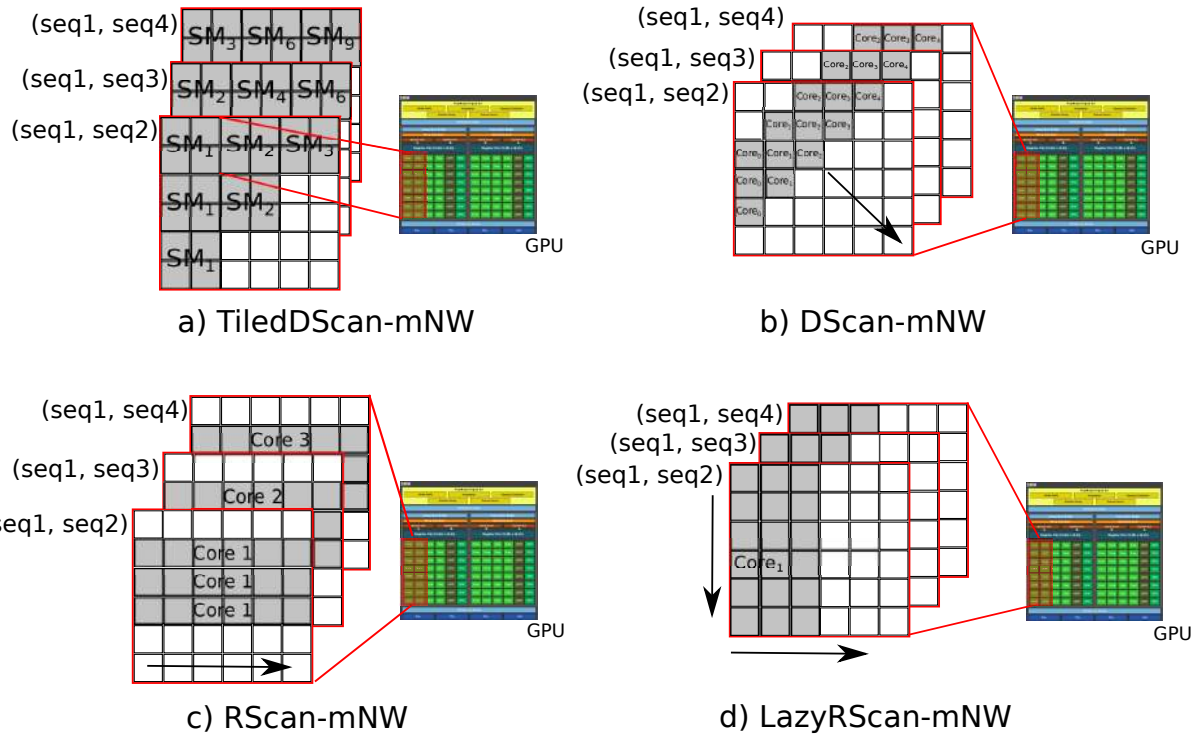


Figura 5.7: Ilustração das 4 implementações em GPU do algoritmo de Needleman e Wunsch, mapeando dentro do núcleo da GPU e SM. No *TiledDScan-mNW*, a matriz de programação dinâmica é fatiadas e cada fatia é processada por um bloco de *threads* e mapeadas dentro do SM. No *DScan-mNW*, cada alinhamento da matriz é processado por um bloco de *threads* e mapeada dentro do SM. No *RScan-mNW* e *LazyRScan-mNW* cada alinhamento da matriz é processado por uma *thread* e mapeada no núcleo. Contudo no *LazyRScan-mNW* cada *thread* somente escreve na memória global o último valor de cada fatia [TLS+14].

O terceiro método *RScan-mNW* é muito similar a implementação paralela inter-tarefa do algoritmo de Smith-Waterman proposto por Liu [LSM09a] e usa granulosidade fina, mapeando a matriz de alinhamentos na GPU, executando cada alinhamento em um simples *kernel* da GPU em linha, conforme mostra a Figura 5.7 (c). Este método utiliza memória compartilhada, permitindo reuso de dados e minimizando transações na memória global, devido ao fato que cada transação possui apenas duas linhas da matriz de alinhamentos na memória compartilhada, uma com valores já calculados e outra com novos

valores a calcular. Desta forma os valores calculados anteriormente são descartados e os novos valores são mantidos em *cache* para próxima iteração. O *kernel* possui duas fases: comunicação e cálculo. Na fase de cálculo, as *threads* dentro do bloco de *threads* operam independentemente, calculando cada linha da matriz de alinhamento e armazenando o resultado na memória compartilhada. Na fase de comunicação, *threads* pertencentes ao mesmo bloco de *threads* cooperam para transferir os dados da linha da matriz de alinhamento da memória compartilhada para memória global. Caso as sequências alinhadas possuírem tamanho muito grande, estas são divididas em fatias de tamanho k, sendo este valor escolhido pelo programador. Fatias muito grande necessitam de maior quantidade de memória compartilhada, que por sua vez limita o número de *threads* ativas em cada *SM*. Por outro lado uma fatia muito pequena leva a baixa utilização de comunicação, prejudicando o desempenho. A principal desvantagem de paralelização deste método é a limitação de memória da GPU, quando é realizado o alinhamento de sequências de tamanho muito grande.

Algoritmo 4: Algoritmo LazyRScan

```
1  Inicialize matriz (w,1);
2  foreach fatia (k,1) do
3      Calcule as pontuações para linha 0;
4      for linha ← 1 to l do
5          Leia a pontuação mais a esquerda da coluna 0 da memória global;
6          for coluna ← 1 to k do
7              Calcule o valor para (linha,coluna) na memória compartilhada;
8          Salve a pontuação mais a direita da coluna 1 na memória global;
9      Avance para próxima fatia k;
10 Retorne as pontuações finais da matriz;
```

Devido ao fato que várias aplicações realizarem o *traceback* somente em um subconjunto de pares de sequências, onde a pontuação seja maior que um determinado valor, foi desenvolvido o quarto e último método *LazyRScan-mNW* para filtrar operações de *traceback* desnecessárias. Este método descarta seguramente informações em uma fatia da matriz de alinhamento durante seu processamento. Sendo extremamente econômico no uso da memória global, evitando seu acesso quando possível.

O *LazyRScan-mNW* divide a matriz de alinhamento em fatias verticais com k colunas conforme mostra a Figura 5.7 (d) e o Algoritmo 4. Este algoritmo armazena somente as pontuações na memória global da última coluna de cada fatia, as demais pontuações são descartadas. Com uma boa escolha de k

é possível reduzir o número de escritas na memória. O processamento de uma fatia de largura k e altura l requer $2l$ operações na memória compartilhada (l leitura e l escrita). Limitando o uso de memória global é possível que mais blocos de *threads* sejam executados em paralelo, ocultando assim a latência da memória global.

Com o *LazyRScan-mNW*, foi possível realizar o alinhamento na ordem de 4.000 a 6.000 sequências por segundo, utilizando GPUs variando de Quadro 2000 4 SM x 48 cores com 1 GB memória global e Tesla K20 13 SM x 192 cores com 4.7 GB memória global, respectivamente.

5.2.7 BLAST

BLAST (do inglês, *Basic Local Alignment Search Tool*, ferramenta básica para pesquisa de alinhamentos locais) [AGM+90] é um heurística para o alinhamento de várias sequências que busca otimizar a medida de similaridade, permitindo a criação de um *tradeoff* entre a velocidade do algoritmo e a sensibilidade por meio de um parâmetro *T*. Alterando o valor de *T* para um valor alto é aumentado a velocidade de execução da ferramenta, porém aumenta a probabilidade de regiões de baixa similaridade serem descartadas.

A ideia principal do algoritmo BLAST é que alinhamentos estatisticamente relevantes possuem um par de sequências idênticas alinhadas com alta pontuação. O BLAST é executado em três passos. De forma resumida, no primeiro passo é realizado o processamento no banco de dados por sequências que possuam pontuação igual ou maior que T, quando alinhadas com uma sequência de busca. Qualquer par de sequências alinhadas que satisfaça esta condição é chamado de *hit*.

No segundo passo o algoritmo de BLAST avalia se cada um dos *hits* estão contidos em alinhamento com pontuação alta o suficiente para serem reportados. Isso é feito estendendo-se os *hits* até que a pontuação diminua a um valor X menor que a maior pontuação encontrada até então.

Por fim o algoritmo BLAST utiliza os alinhamentos obtidos para avaliar a sua relevância estatística. Os alinhamentos considerados significantes são chamados de pares de segmento com alta pontuação.

5.2.8 CUDA ClustalW

Hung *et al.* [HLL+15], apresentam uma versão em CUDA do Clustal W, chamada de CUDA ClustalW. Nesta proposta foi utilizada a paralelização *intra-task* em uma ou mais GPU. O CUDA ClustalW, paraleliza somente o primeiro estágio do alinhamento progressivo, pois na análise do autor utilizando 1.000 sequências, observou-se que a cálculo da matriz de distâncias consome 90%

do tempo total para realizar o alinhamento final. Para *k* sequências o CUDA ClustalW realiza $k^2/2$ alinhamento entre pares no passo de cálculo da matriz de distâncias, atribuindo cada par de sequências a um bloco de *threads*. O alinhamento na GPU é realizado utilizando algoritmos de programação dinâmica. O CUDA ClustalW obteve um *speedup* de 33 vezes comparado com a versão serial do Clustal W.

CAPÍTULO

6

Implementação da proposta

A implementação proposta explorou a paralelização do alinhamento de várias sequências nos três estágios e em dois níveis de granulosidade. Na granulosidade grossa todas as tarefas de cálculo são distribuídas no *cluster* entre o nó principal chamado *master* e os nós secundários chamados *slaves* utilizando MPI. Na granulosidade fina as tarefas são calculadas dentro da GPU utilizando CUDA e seus resultados são devolvidos pelo MPI ao nó *master*.

Neste capítulo serão descritos os métodos utilizados para realizar esta paralelização híbrida dos três estágios do alinhamento de várias sequências.

6.1 Distribuição no Cluster

Conforme ilustra a Figura 6.1, a distribuição de processamento para cálculo da matriz de distâncias entre os nós do *cluster* é realizada conforme segue. Inicialmente o nó principal realiza a leitura de um arquivo no padrão FASTA contendo n sequências de aminoácidos. O nó principal calcula a distribuição de tarefas entre os p nós secundários, onde p é quantidade de nós disponíveis para realizar o processamento. Cada nó p realiza $(n(n-1)/2)/p$ alinhamentos, onde n é a quantidade de sequências de entrada. Estas sequências são empacotadas e distribuídas entre p nós secundários utilizando MPI. Este maneira de distribuição de tarefas é a mesma utilizada pelo ClustalW-MPI.

Para ilustrar um exemplo é mostrado na Figura 6.1 a distribuição de 6 sequências de entrada entre o nó principal e os nós secundários. Para calcular a matriz de distâncias é necessário fazer a comparação par-a-par com as sequências de entrada, portanto tem-se 15 tarefas de cálculo que serão

distribuídos entres 4 nós secundários.

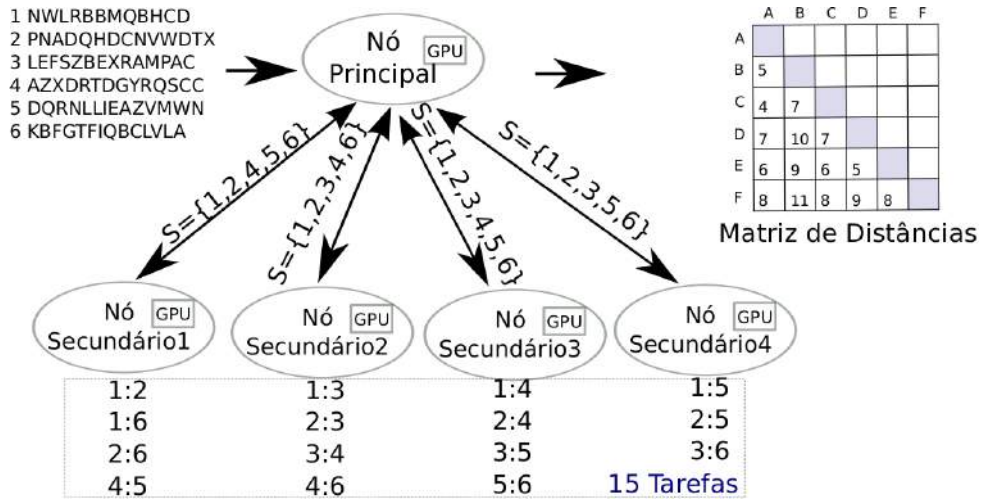


Figura 6.1: Distribuição de várias sequências entre os nós do *cluster*. O nó principal recebe um arquivo FASTA contendo várias sequências, as tarefas são divididas entre os nós secundários e somente as sequências são enviadas. Os nós secundários recebem as sequências e efetuam um cálculo e devolvem as pontuações da matriz de distância que são recebidos e gerenciados pelo nó principal.

Cada nó secundário ao receber seu pacote único de trabalho, ou seja, somente as sequências necessárias para o cálculo, envia seu subconjunto de sequências juntamente com a matriz de pontuação à GPU para o cálculo da matriz de distâncias entre estas sequências, como exemplo o nó secundário 2 recebe as sequências S=$\{1, 2, 3, 4, 6\}$, para realizar 4 tarefas de cálculo.

6.2 Matriz de Distâncias

Após a distribuição de sequências, cada nó secundário dotado de uma GPU realiza o cálculo da matriz de distâncias para seu conjunto de sequências correspondente. A matriz H é calculada utilizando programação dinâmica e fórmula de recorrência proposto por Needleman e Wunsch descrito na Seção 3.1.1.

O cálculo das pontuações entre as sequências é realizado pelo algoritmo *LazyRScan-mNW* [TLS+14] apresentado na Seção 5.2.6 simultaneamente nas GPUs dos p nós secundários. A matriz de distâncias armazenada na memória global da GPU é dividida em fatias, sendo que cada fatia é calculada por um bloco de *threads*.

Especificamente as sequências são armazenadas na memória global e os cálculos são realizados com dados da memória compartilhada. A memória compartilhada contém uma linha híbrida da fatia que esta sendo calculada, composta de duas linhas continua da memória global. É realizado um troca otimizada de informações entre estas memórias, permitindo economizar me-

mória global para realizar os cálculos da matriz de distâncias conforme mostra a Figura 6.2.

Esta otimização permite utilizar mais *threads* por bloco, sendo que cada *thread* requisita somente uma quantidade linear de memória global. Portanto a memória requisitada é igual a *num-par* × l , onde l é o tamanho da maior sequência e *num-par* é o número de pares de alinhamentos a serem calculados em paralelo em uma GPU. Ao invés de ler e gravar na memória global em cada célula calculada, é realizada leitura e escrita na memória global a cada k células, onde k é a largura da matriz de alinhamento. Isso permite esconder à latência de acesso a memória global. A quantidade de memória compartilhada requisitada para armazenar as pontuações é economizada utilizando este método.

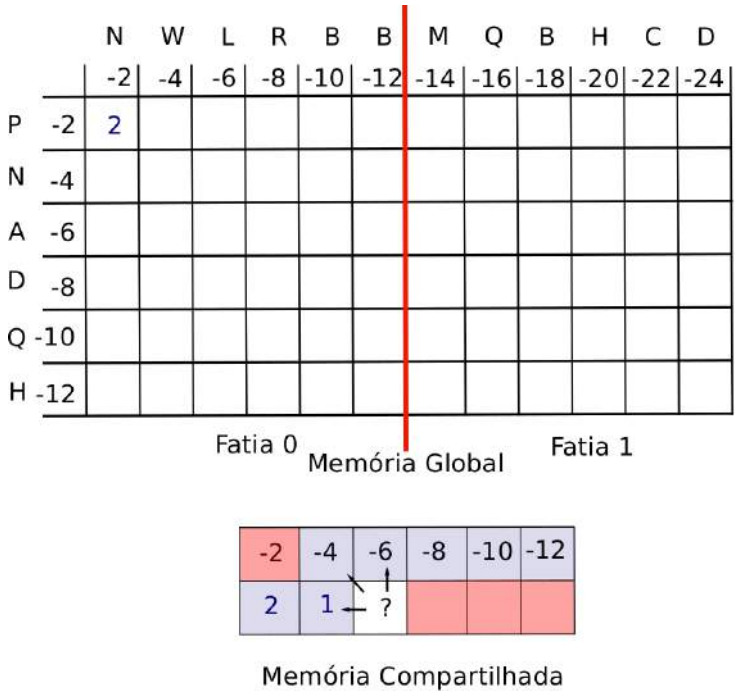


Figura 6.2: Ilustração do cálculo do alinhamento utilizando memória global e memória compartilhada. O cálculo na memória compartilhada (célula em branco) ocorre da esquerda para direita na horizontal, as células obsoletas (células em vermelho) são gradualmente descartadas da memória compartilhada.

Após o cálculo na GPU todas as pontuações são copiadas para CPU e posteriormente devolvidas via MPI ao nó principal. A complexidade de espaço para realizar os cálculos da matriz de distâncias é $O(n^2)$ para alinhar duas sequências de tamanho *n*.

6.3 Árvore Guia

O segundo estágio do alinhamento de várias sequências, construção da árvore guia, foi implementado com base no método NJ (do inglês, *Neighbor-*

Joining, Junção de Vizinho), proposto por Saitou e Nei [SN87], conforme Algoritmo 5.

A Figura 6.3 ilustra os passos de execução do algoritmo *Neighbor-Joining*. Cada passo foi detalhado de forma a representar a matriz de distâncias de entrada e a árvore sem raiz produzida.

Este estágio do alinhamento de várias sequências foi paralelizado em uma única GPU, conforme ilustra a Figura 6.5, devido a grande dependência de dados requerido pelo método *Neighbor-Joining*, dificultando a divisão da matriz D_{nn} de n sequências em mais de uma GPU. Ao término do primeiro estágio, o nó principal preenche a matriz de distâncias D_{nn}.

Algoritmo 5: Algoritmo Neighbor-Joining

1. Baseado na matriz de distâncias D_{nn} calcule: $S_{ij} = (n-2)D_{ij} - R_i - R_j$, onde $R_i = \sum_{k=1}^{n} D_{ik}$ e $R_j \sum_{k=1}^{n} D_{jk}$
2. Selecione o menor $M_{ij} = D_{ij} - S_i - S_j$.
3. Crie o novo nó u e calcule o tamanho dos ramos para o novo nó u tal que $D_{iu} = 1/(2(n-2))[(n-2)D_{ij} + R_i - R_j]$ e $D_{ju} = 1/(2(n-2))[(n-2)D_{ij} - R_i + R_j]$
4. Exclua i e j de D e adicione u
5. Calcule a nova distância de cada sequência para o novo nó u tal que $D_{uk} = 1/2[D_{fk} + D_{gk} - D_{fg}]$, onde k é o nó que desejamos calcular a distância para f e g que são membros dos pares juntados
6. Inicie o algoritmo novamente, substituindo o par vizinho unido (novo nó) e usando as distâncias calculadas no passo anterior

Conforme ilustra a Figura 6.4, uma matriz de distância de tamanho $n \times n$, e um bloco de células de tamanho $m \times m$ desta matriz, o número total de blocos de células é definido pela Equação 6.1.

$$totalBlocos = \left(\frac{blocos \times (blocos + 1)}{2} \right), blocos = (n + m - 1)/m \tag{6.1}$$

De acordo com [Coo12], o número de *threads* por bloco de *threads* deve ser múltiplo do número de *warps* (consistindo de 32 *threads*). Desta forma, na implementação proposta, foi escolhido 512 *threads* por bloco, considerando o número de registradores por bloco disponível, para ter um bom equilíbrio de carga, desta forma definiu-se o número máximo de *threads* por bloco.

Com a matriz de distâncias D_{nn} na memória da CPU, a implementação proposta lança um *kernel* para o passo 1 e um *kernel* para o passo 2 do algoritmo de *Neighbor-Joining*. No primeiro passo, para o cálculo da matriz de divergência a matriz D_{nn} é copiada para GPU e são lançados $n/MAX\_THREADS$ blocos, e n *threads* por bloco. Após cálculo da matriz de divergência, a matriz

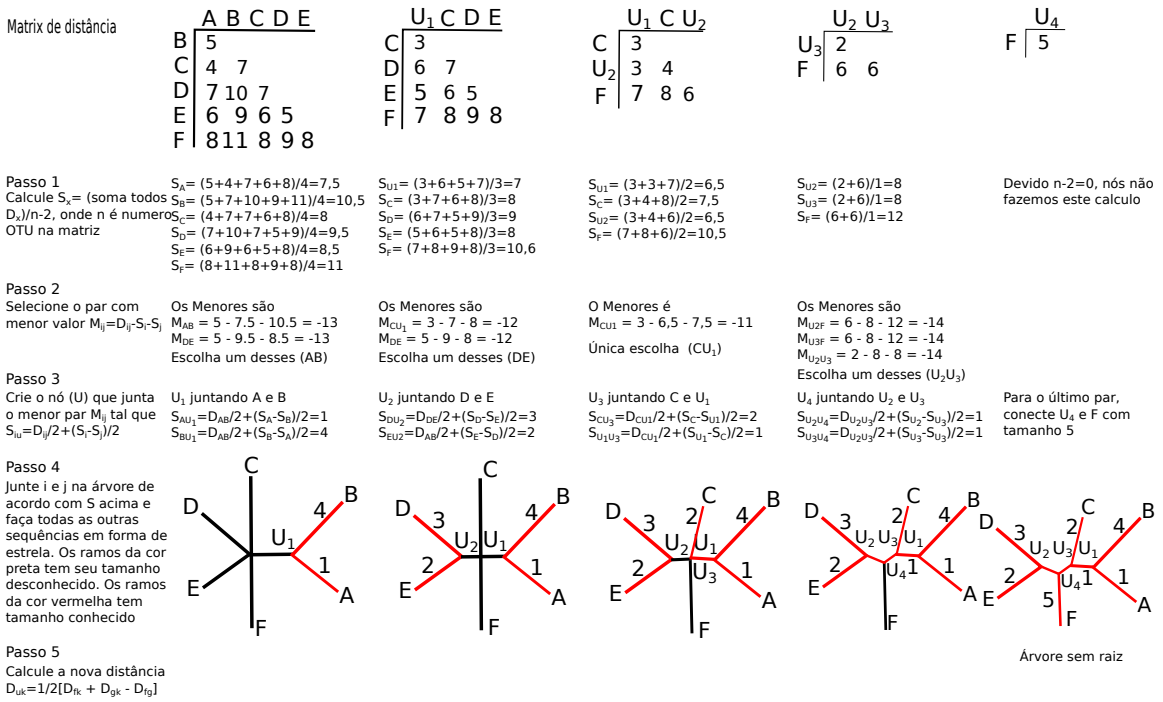


Figura 6.3: Exemplo de execução do algoritmo *Neighbor-Joining*, com seis sequências. A entrada é a matriz de distâncias e a saída uma árvore sem raiz.

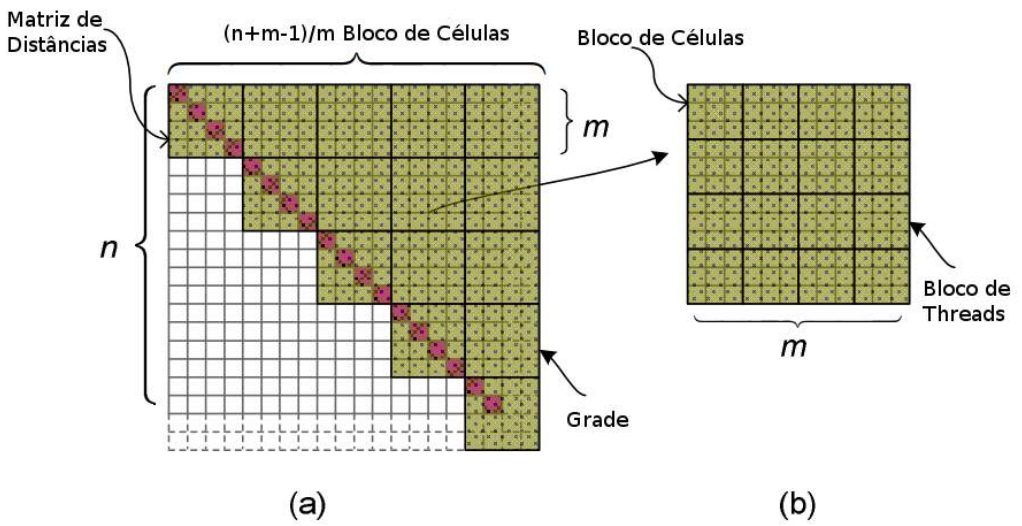


Figura 6.4: Método simples e direto de mapeamento da matriz de distância na GPU. a) a matriz de distância é mapeada para um *grid* de bloco de *threads*. b) um bloco de células corresponde a um bloco de *threads*. [LSM09b]

S_{ij} é copiada para CPU. No segundo passo, para encontrar o menor elemento, utiliza-se uma operação de redução em CUDA para encontrar o menor valor em uma matriz bidimensional, sendo o número de blocos e a quantidade de *threads* por blocos, calculados dinamicamente de acordo com a quantidade de sequências de entrada. A matriz S_{ij} é copiada para GPU e um segundo *kernel* é lançado. O resultado do segundo passo é devolvido para CPU, sendo os índices i e j utilizados na CPU para criar o novo nó e realizar o cálculo dos tamanhos dos ramos para o novo nó e cálculo da nova distância de cada

sequência para o novo nó (passo 3-5) do algoritmo de *Neighbor-Joining*. A cada iteração do algoritmo a árvore é criada na CPU, e ao término da execução do algoritmo é salva em disco no formato PHYLIP (do inglês, *Phylogeny Inference Package*, Pacote de Inferência Filogenética).

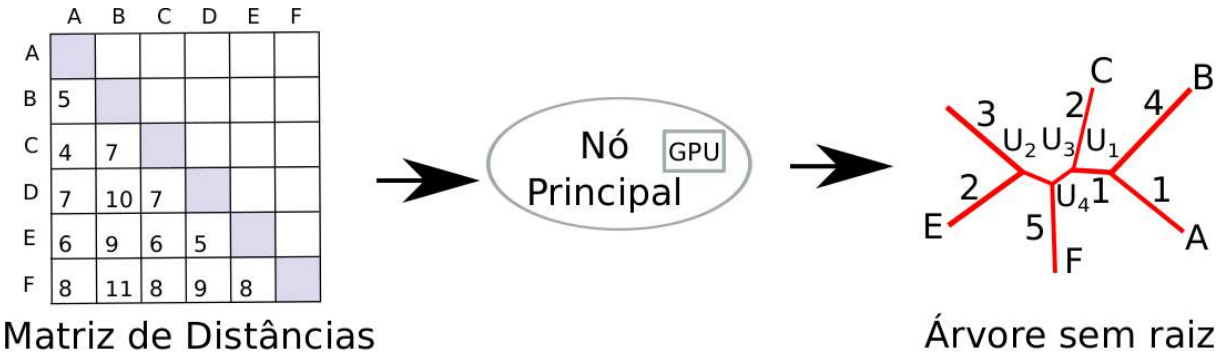


Figura 6.5: Execução do Algoritmo de *Neighbor-Joining* pela implementação proposta utilizando apenas uma GPU. A árvore guia é calculada em uma GPU

6.4 Alinhamento Progressivo

A implementação proposta para o último estágio do alinhamento de várias sequências, é uma extensão do ClustalW-MPI [Li03], com algumas alterações. O ClustalW-MPI executa este passo utilizando apenas MPI, a implementação proposta acrescenta o uso de GPU seguindo os seguintes passos conforme é mostrado a Figura 6.6.

Com a árvore guia em memória na CPU é realizado o cálculo de similaridade entre as sequências, ou seja, é realizado um agrupamento das sequências de acordo com a pontuação na árvore, obtendo assim uma árvore guia com raiz, o qual guia a ordem de execução dos alinhamentos. As sequências menos divergentes são alinhadas inicialmente, somente depois as sequências mais divergentes são alinhadas.

Um vetor $dest[]$ é utilizado para controlar a distribuição de trabalho para os nós escravos. Sendo $dest[0]$ utilizado para armazenar a quantidade de processadores disponíveis e $dest[1]..dest[p]$ o *rank* dos p processadores disponíveis.

No laço principal cada sequência do conjunto agrupado é enviado utilizando MPI para um processador p disponível. Caso não tenha processador disponível o processador p principal fica aguardando retornar resultados. Obedecendo a dependência dos nós na árvore guia, cada processador p escravo recebe um nó da árvore e copia suas sequências para a GPU, alinhando-as e devolvendo via MPI no vetor global $seq\_array[][]$ da CPU principal. O alinhamento executado na GPU utiliza o mesmo algoritmo do ClustalW-MPI.

Apenas como ilustração, é possível observar na Figura 6.6 que em uma primeira execução do algoritmo, os nós U_1 e U_2 da árvore guia são executados

ao mesmo tempo em GPUs diferentes. Somente após a primeira execução do laço principal, U_3 é executado em seguida, devido a dependência com U_1. Na próxima execução do laço principal U_4 é alinhado e por fim U_5 tem seu alinhamento finalizado pela GPU.

Ao término de execução do laço principal tem-se como saída o resultado do alinhamento de várias sequências.

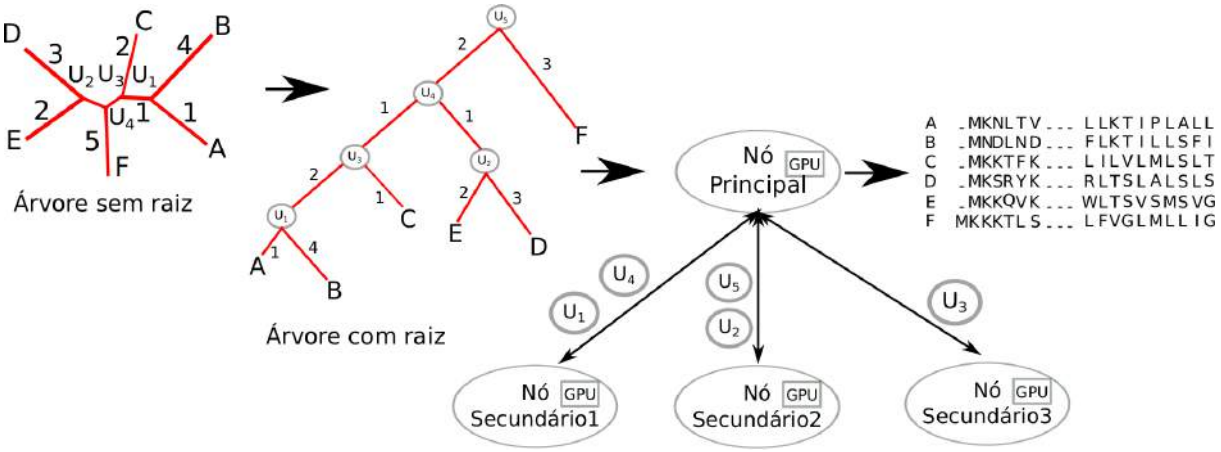


Figura 6.6: Alinhamento progressivo da implementação proposta. A árvore guia é utilizada para definir a ordem dos alinhamentos, sendo os nós U_1 e U_2 alinhados ao mesmo tempo. U_3,U_4 e U_5 são alinhados em seguida devido a dependência de execução.

A implementação proposta para realizar o alinhamento de várias sequências foi escrita para ser utilizada em nós que possuam apenas uma GPU, contudo com alguns ajustes é possível utilizar mais de uma GPU por nó, de forma a possibilitar um melhor balanceamento de carga.

CAPÍTULO

7

Resultados

Neste capítulo apresentam-se uma comparação de resultados obtidos com a implementação proposta e o ClustalW-MPI [Li03]. Escolheu-se este alinhador de várias sequências devido sua popularidade e quantidade de citações na literatura. Utilizamos a medida de tempo para comparar ambos os métodos, e na entrada utilizamos a base de dados do HIV-1 [LSM09a], sendo estes casos de teste especificados a seguir:

Caso 01: 400 sequências de tamanho médio 851;

Caso 02: 1000 sequências de tamanho médio 851;

Caso 03: 2000 sequências de tamanho médio 234;

Caso 04: 4000 sequências de tamanho médio 238;

Caso 05: 4000 sequências de tamanho médio 61;

Caso 06: 8000 sequências de tamanho médio 68;

7.1 Ambiente de testes

Os casos de teste foram executados em um *cluster* contendo 44 nós com processadores Intel(R) Xeon(R) CPU X3440@2.53GHz de 4 núcleos, 4 GB de memória RAM, sistema operacional Rocks 6.2 e a comunicação entre eles é feita através de placas Myrinet-10G, sendo 6 nós equipado com uma placa GPUs Nvidia Quadro 600. Cada GPU possui 1 Gb de memória com 2 *Streaming Multiprocessors (SMs)* e 48 *Streaming Processors (SPs)* em cada SM, totalizando 96 CUDA cores.

7.2 Desempenho dos métodos

Para analisar o desempenho dos métodos propostos, inicialmente foi aplicado os casos de teste no ClustalW-MPI. Mostra-se na Tabela 7.1, as medidas de tempo de execução dos três estágios no ClustalW-MPI lançando 4 processos por nó, opção disponível no MPI. Foi observado que utilizando 4 processos por nó o ClustalW-MPI obteve um melhor desempenho de tempo, utilizando uma quantidade maior de processos por nó houve um aumento no tempo total de execução, devido ao *overhead* de comunicação MPI entre os processos.

Caso de teste	Número de Nós					
	1	2	4	8	16	32
Caso 01	42,9	18,4	8,7	4,4	2,3	1,3
Caso 02	264,9	113,9	53,5	26,4	13,4	7,1
Caso 03	59,2	26,8	14,1	8,5	5,7	4,4
Caso 04	262,4	125,1	70,7	46,1	34,5	28,8
Caso 05	31,7	25,4	23,2	22,1	22,7	21,4
Caso 06	236,3	205,2	194,1	188,6	186,2	185,4

Tabela 7.1: Resultados obtidos, em minutos, da execução do *ClustalW-MPI* utilizando 1,2,4,8,16 e 32 nós.

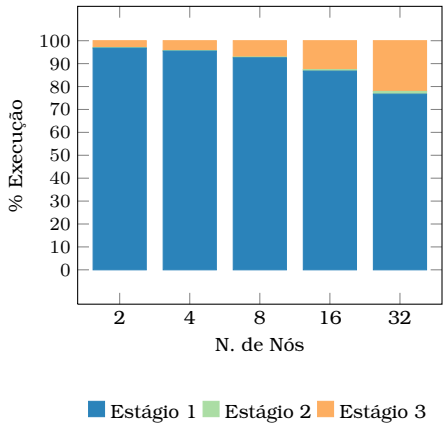
Ainda no ClustalW-MPI foi observado os tempos de execução de cada um dos três estágios. É mostrado na Figura 7.1 seu percentual de tempo de execução, ou seja, qual a porcentagem de tempo que cada estágio necessita para concluir com os casos de teste em estudo. Analisando a Figura 7.1a, pode-se observar que o primeiro estágio necessita em média 90% do tempo total de execução no Caso 01, quando a quantidade de sequências é menor. A medida que aumentamos a quantidade de sequências a serem alinhadas o segundo estágio consome mais tempo, chegando a gastar em média 92% do tempo total como mostra o Caso 06 na Figura 7.1f.

Conforme dados apresentados por Liu [LSM09b], a complexidade do Clustal W descrita na Tabela 5.1 é muito similar ao ClustalW-MPI, levando a seguinte conclusão com relação ao tempo de execução:

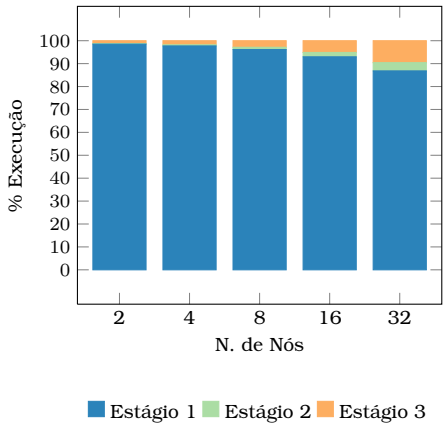
- O estágio 1 sempre demora mais do que o estágio 3;
- Se $n > l_{ave}$ então o estágio 2 demora mais que o estágio 3;
- Se $n > l_{ave}^2$ então o estágio 2 demora mais que o estágio 1.

onde n é a quantidade de sequências a serem alinhadas e l_{ave} é o tamanho médio de todas as sequências.

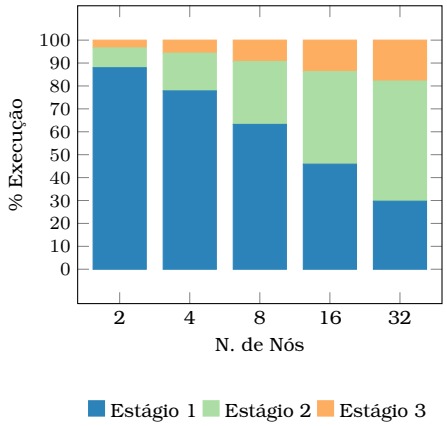
Com objetivo de detalhar os resultados, foi aplicado os casos de teste na implementação proposta, sendo analisado o primeiro estágio do alinhamento



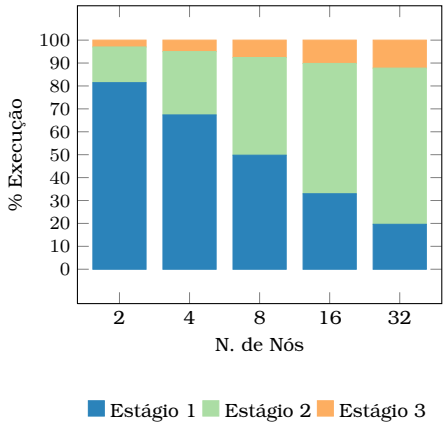
(a) Caso 01 - 400 sequências de tamanho médio 851



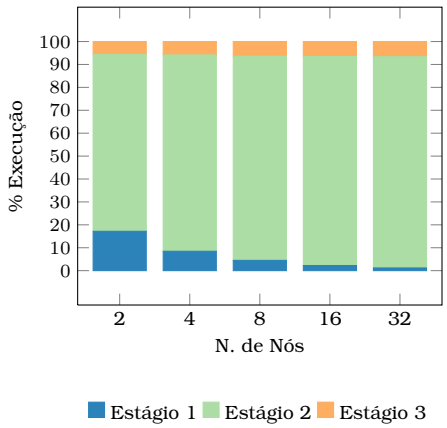
(b) Caso 02 - 1000 sequências de tamanho médio 851



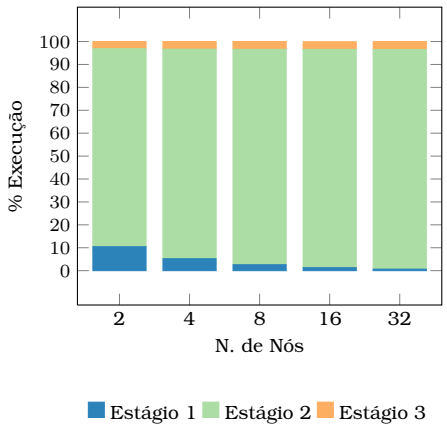
(c) Caso 03 - 2000 sequências de tamanho médio 234



(d) Caso 04 - 4000 sequências de tamanho médio 238



(e) Caso 05 - 4000 sequências de tamanho médio 61



(f) Caso 06 - 8000 sequências de tamanho médio 68

Figura 7.1: Porcentagem de tempo de execução do ClustalW-MPI.

de várias sequências, que é a construção da matriz de distâncias, ou seja, o cálculo da pontuação entre pares de todas as sequências de entrada utilizando programação dinâmica em GPU. Na Tabela 7.2 e Figura 7.2, observa-se as medidas de tempo em segundos da implementação paralela proposta com 1,2,4 e 6 GPUs.

Comparando a implementação proposta com o ClustalW-MPI, pode-se examinar um ganho para casos de teste inferior a 4.000 sequências de tamanho médio 238, obtendo um *speedup* de até 60 vezes conforme Tabela 7.3 e Figura 7.3. Acima de 4.000 sequências de tamanho médio 61 com a quantidade de GPUs utilizadas, não foi observado ganho em relação ao ClustalW-MPI.

Caso	Número de GPUs			
	1	2	4	6
Caso 01	4,7	2,2	1,1	1,0
Caso 02	39,0	19,3	9,5	8,0
Caso 03	125,5	60,8	16,9	9,5
Caso 04	689,8	169,3	85,2	53,4
Caso 05	421,7	122,0	59,4	42,6
Caso 06	1167,3	604,5	302,4	214,5

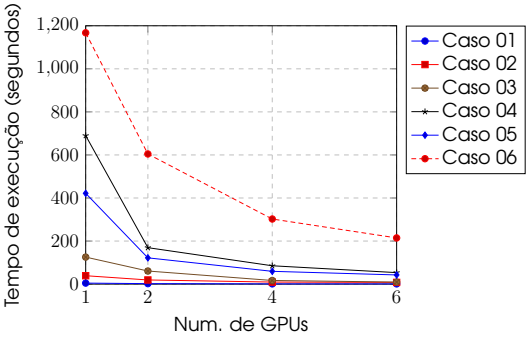


Tabela 7.2 e Figura 7.2: Tempo de execução, em segundos, do primeiro estágio calculado pela implementação proposta utilizando 1,2,4 e 6 GPUs.

Caso de teste	Número de GPUs			
	1	2	4	6
Caso 01	12,6	27,1	53,7	60,8
Caso 02	9,6	19,3	39,1	46,5
Caso 03	0,6	1,3	4,7	8,3
Caso 04	0,5	2,0	4,0	6,4
Caso 05	0,0	0,1	0,3	0,4
Caso 06	0,1	0,1	0,2	0,3

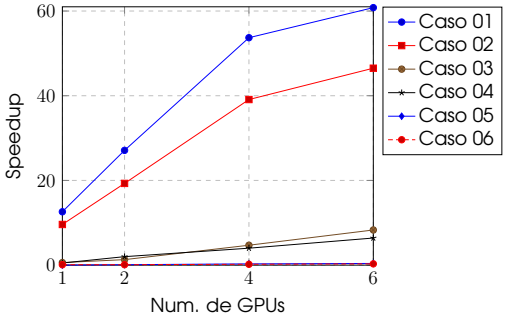


Tabela 7.3 e Figura 7.3: *Speedup* da implementação proposta em relação ao ClustalW-MPI referente ao primeiro estágio, utilizando 1,2,4 e 6 GPUs.

Seguindo o detalhamento dos resultados apresentados pela implementação proposta, observa-se na Tabela 7.4 os tempos de execução e *speedup* dos casos de teste utilizando 1 GPU para construção da árvore guia, conforme descrito no Capítulo 6. É possível observar que a implementação proposta não apresenta um comportamento decrescente com relação ao tempo de execução como se pode observar no primeiro estágio, devido a utilização de uma única GPU.

O ClustalW-MPI possui uma opção para executar o segundo estágio em paralelo utilizando MPI. Em testes realizados notou-se um desempenho inferior a execução serial, desta forma comparamos a execução serial do ClustalW-MPI utilizando 32 nós, com a implementação proposta em uma única GPU, observando uma melhora de performance a medida que a quantidade de sequências aumenta, chegando a um *speedup* de 3 vezes.

Para finalizar o detalhamento do alinhamento de várias sequências na implementação proposta, os resultados de tempo de execução podem ser obser-

Caso de teste	Resultados	
	tempo	Speedup
Caso 01	1,1	0,8
Caso 02	10,4	1,5
Caso 03	63,7	2,2
Caso 04	452,8	2,6
Caso 05	452,8	2,6
Caso 06	3415,7	3,1

Tabela 7.4: Resultados obtidos, do segundo estágio calculado pela implementação proposta utilizando 1 GPUs.

vados na Tabela 7.5 e Figura 7.4. Constata-se que o desempenho deste estágio varia dentro de uma faixa de tempo. Ao contrário da segunda fase que utiliza somente uma GPU, este estágio utiliza 1,2,4 e 6 GPUs.

Comparando o terceiro estágio da implementação proposta com o ClustalW-MPI, pode-se observar na Tabela 7.6 e Figura 7.5 um *speedup* variando entre 5 a 20 vezes para os casos de testes em estudo. Observa-se ainda um desempenho melhor para entrada com quantidades inferior a 4.000 sequências de tamanho médio 238, da mesma forma observada no primeiro estágio.

Caso de teste	Número de GPUs			
	1	2	4	6
Caso 01	2,0	1,4	1,4	1,2
Caso 02	2,8	2,3	2,9	3,1
Caso 03	3,7	2,1	2,7	2,2
Caso 04	6,2	5,6	5,6	6,0
Caso 05	14,1	13,5	14,8	15,9
Caso 06	41,6	43,1	41,8	42,2

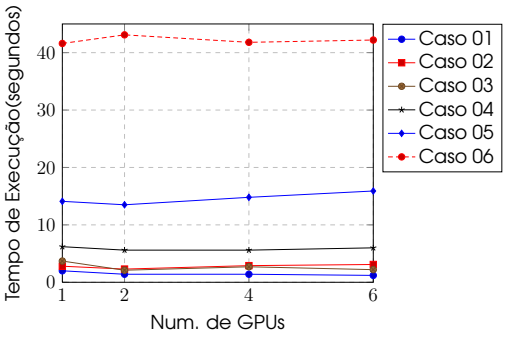


Tabela 7.5 e Figura 7.4: Tempo de execução, em segundos, do terceiro estágio calculado pela implementação proposta utilizando 1,2,4 e 6 GPUs.

Caso de teste	Número de GPUs			
	1	2	4	6
Caso 01	8,6	12,2	12,8	14,1
Caso 02	14,4	17,5	13,9	13,3
Caso 03	12,7	22,7	17,8	21,6
Caso 04	33,8	37,3	37,3	35,1
Caso 05	5,9	6,2	5,6	5,3
Caso 06	9,1	8,8	9,1	9,0

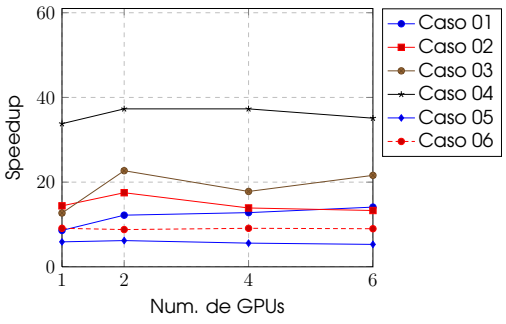


Tabela 7.6 e Figura 7.5: *Speedup* da implementação proposta em relação ao ClustalW-MPI referente ao terceiro estágio, utilizando 1,2,4 e 6 GPUs.

Após analisar os três estágios individualmente, pode ser observado na Tabela 7.7 e Figura 7.6, o tempo total gasto pela implementação proposta para

Caso de teste	Número de GPUs			
	1	2	4	6
Caso 01	0,1	0,1	0,1	0,1
Caso 02	0,9	0,5	0,4	0,4
Caso 03	3,2	2,1	1,4	1,3
Caso 04	19,1	10,5	9,1	8,5
Caso 05	14,8	9,8	8,8	8,5
Caso 06	71,1	67,7	62,7	61,2

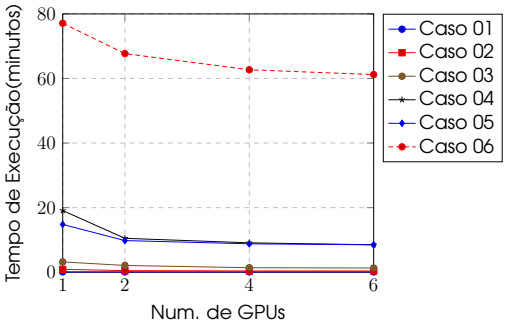


Tabela 7.7 e Figura 7.6: Resultados obtidos, em minutos, pela implementação proposta utilizando 1,2,4 e 6 GPUs, para cálculo total do alinhamento de várias sequências.

Caso de teste	Número de GPUs			
	1	2	4	6
Caso 01	9,9	16,4	21,6	23,3
Caso 02	8,2	13,4	18,7	19,9
Caso 03	1,4	2,1	3,2	3,5
Caso 04	1,5	2,8	3,2	3,4
Caso 05	1,4	2,2	2,4	2,5
Caso 06	2,4	2,7	3,0	3,0

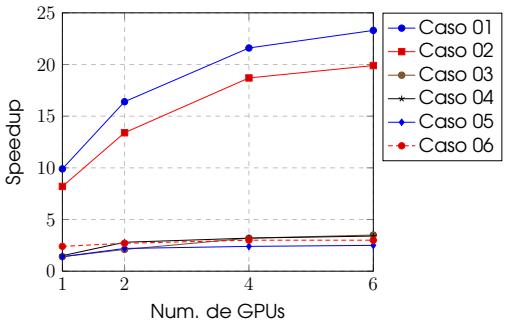


Tabela 7.8 e Figura 7.7: *Speedup* da implementação proposta em relação ao ClustalW-MPI, utilizando 1,2,4 e 6 GPUs.

executar os casos de teste em estudo. Utilizou-se a medida de tempo em minutos, com 1,2,4 e 6 GPUs.

A comparação geral entre a implementação proposta e o ClustalW-MPI é mostrado na Tabela 7.8 e Figura 7.7. É possível observar que a implementação proposta obteve um resultado melhor no alinhamento com quantidade inferior a 1.000 sequências de tamanho médio 851, alcançando um *speedup* de até 23 vezes. Para quantidades entre 1.000 sequências de tamanho médio 851 a 8.000 sequências de tamanho médio 68, a implementação proposta atingiu um *speedup* de até 3 vezes. Considerando as implementações existentes que paralelizam os três estágios do alinhamento de várias sequências, a implementação proposta é mais rápida.

Apesar de soluções propostas mais recentes para o alinhamento de várias sequências utilizando GPU [HLL+15] [LSM09a] [LSVMW07] e MPI [SKP09], foi realizado somente a comparação com o ClustalW-MPI devido sua popularidade e pela quantidade de citações na literatura e também por algumas propostas não disponibilizarem seus fontes para testes.

Nos experimentos realizados não foi fixado o tamanho das sequências de modo observar a relação entre a quantidade de sequências e o seu tamanho médio. Também não foi realizado um comparativo com a qualidade dos alinhamentos produzidos pela implementação proposta, devido a escolha inicial

da medida de tempo para comparar ambos os métodos.

CAPÍTULO

8

Conclusão e trabalhos futuros

O alinhamento de várias sequências biológicas é uma das ferramentas muito importante para o estudo de problemas biológicos, sendo utilizado para inferir informações sobre espécies em estudo. O problema de alinhar várias sequências é NP-Difícil, conforme foi abordado neste trabalho, sendo o tempo de solução de alto custo para uma quantidade muito grande de sequências.

Vários algoritmos e métodos têm sido propostos para resolver este problema, utilizando abordagens exatas, aproximadas e até mesmo heurísticas. Aliados a computação de alto desempenho, existem diversas propostas para acelerar a execução desta tarefa. Algumas propostas paralelizam apenas o primeiro estágio do alinhamento de várias sequências, outras paralelizam apenas o segundo estágio. Além das paralelizações ocorrerem em apenas um determinado estágio, normalmente utilizam apenas uma técnica, como por exemplo, o ClustalW-MPI que utiliza somente MPI ou MSA-CUDA que utiliza somente CUDA. Poucas são as propostas que utilizam paralelização híbrida em *cluster*, um exemplo foi apresentado na seção 5.2.6.

Foi proposto neste trabalho uma heurística em *cluster*, utilizando MPI e CUDA, para realizar a paralelização nos três estágios do alinhamento de várias sequências. Os *speedup* com relação ao ClustalW-MPI ficaram entre 3 e 23 vezes, para 8.000 sequências de tamanho médio 68 e 1.000 sequências de tamanho médio 851, respectivamente.

Conforme resultados apresentados, o tempo gasto para concluir cada um dos estágios varia de acordo com a quantidade e tamanho das sequências fornecida como entrada. Com uma quantidade menor de sequências, o primeiro estágio consome mais tempo para ser concluído, quando aumentamos a quantidade de sequências o segundo estágio torna-se mais demorado.

Em uma implementação futura, caso necessite realizar o alinhamento de muitas sequências sugere-se foco no segundo estágio. A implementação proposta utiliza somente uma GPU para este estágio, é possível realizar algumas otimizações de memória, balanceamento de carga e ajustes na quantidade de blocos/*threads*, além de realizar estudo aprofundado dos métodos atuais e até mesmo novos métodos, com isso espera-se alcançar um desempenho de tempo melhor. Entretanto não basta acelerar a execução do alinhamento, faz-se necessário também utilizar algum *benchmark* com por exemplo o BAliBASE [TKRP05], para verificar a qualidade do alinhamento. Ainda para trabalhos futuros é possível estudar a utilização de um *cluster* com máquinas heterogêneas com mais de uma GPU.

Referências Bibliográficas

[AGM+90] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers e David J Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990. Citado na página 43.

[AL89] Stephen F Altschul e David J Lipman. Trees, stars, and multiple biological sequence alignment. *SIAM Journal on Applied Mathematics*, 49(1):197–209, 1989. Citado na página 38.

[Ara12] Francisco Eloi Soares de Araujo. *Alinhamentos e comparação de sequências*. Tese de Doutorado, Universidade de São Paulo, 2012. Citado nas páginas xvii e 13.

[BCC+13] Dennis A Benson, Mark Cavanaugh, Karen Clark, Ilene Karsch-Mizrachi, David J Lipman, James Ostell e Eric W Sayers. Genbank. *Nucleic acids research*, 41(D1):D36–D42, 2013. Citado na página 1.

[BDV01] Paola Bonizzoni e Gianluca Della Vedova. The complexity of multiple sequence alignment with SP-score that is a metric. *Theoretical Computer Science*, 259(1):63–79, 2001. Citado nas páginas 2 e 15.

[Bro06] Terence A Brown. *Genomes*. Garland science, 2006. Citado nas páginas xiii, xvii, 5, 6, 7, e 8.

[CHLW05] Francis YL Chin, NL Ho, Tak Wah Lam e Prudence WH Wong. Efficient constrained multiple sequence alignment with performance guarantee. *Journal of Bioinformatics and Computational Biology*, 3(01):1–18, 2005. Citado na página 17.

[CL88] Humberto Carrillo e David Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988. Citado nas páginas 1, 2, e 19.

[Coo12] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Elsevier, 2012. Citado nas páginas xiii, xiv, 25, 26, 27, 29, e 48.

[CWC92] SC Chan, AKC Wong e DKY Chiu. A survey of multiple sequence comparison methods. *Bulletin of mathematical biology*, 54(4):563–598, 1992. Citado nas páginas 15 e 16.

[CYK+11] Eunice C Chen, Shigeo Yagi, Kristi R Kelly, Sally P Mendoza, RP Tarara, DR Canfield, N Maninger, A Rosenthal, A Spinner, KL Bales et al. Cross-species transmission of a novel adenovirus associated with a fulminant pneumonia outbreak in a new world monkey colony. *PLoS Pathog*, 7(7):e1002155, 2011. Citado nas páginas xiii e 14.

[Dah05] Ralf Dahm. Friedrich miescher and the discovery of DNA. *Developmental biology*, 278(2):274–288, 2005. Citado na página 5.

[dB03] Rogério Theodoro de Brito. Alinhamento de seqüências biológicas. Dissertação de Mestrado, Universidade de Sao Paulo, 2003. Citado nas páginas 1, 21, e 22.

[DEKM98] Richard Durbin, Sean R Eddy, Anders Krogh e Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998. Citado nas páginas xiii e 13.

[ES04] Robert C Edgar e Kimmen Sjölander. A comparison of scoring functions for protein sequence profile alignment. *Bioinformatics*, 20(8):1301–1308, 2004. Citado na página 9.

[FD87] Da-Fei Feng e Russell F Doolittle. Progressive sequence alignment as a prerequisitetto correct phylogenetic trees. *Journal of molecular evolution*, 25(4):351–360, 1987. Citado na página 22.

[GLS99] William Gropp, Ewing Lusk e Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999. Citado nas páginas 25 e 29.

[Gus93] Dan Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55(1):141–154, 1993. Citado nas páginas 1, 23, e 38.

[Gus97] Dan Gusfield. Algorithms on strings, trees, and sequences. *Computer Science and Computional Biology (Cambrigde, 1999)*, 1997. Citado na página 38.

[Hag97] Torben Hagerup. Allocating independent tasks to parallel processors: an experimental study. *Journal of Parallel and Distributed Computing*, 47(2):185–197, 1997. Citado na página 35.

[HEGMG08] Manal Helal, Hossam El-Gindy, Lenore Mullin e Bruno Gaeta. Parallelizing optimal multiple sequence alignment by dynamic programming. Em *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*, páginas 669–674. IEEE, 2008. Citado na página 31.

[HH92] Steven Henikoff e Jorja G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992. Citado na página 12.

[HLL+15] Che-Lun Hung, Yu-Shiang Lin, Chun-Yuan Lin, Yeh-Ching Chung e Yi-Fang Chung. CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on Multi-GPUs. *Computational Biology and Chemistry*, 2015. Citado nas páginas 43 e 58.

[HS88] Desmond G Higgins e Paul M Sharp. CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988. Citado nas páginas 3 e 33.

[JEP10] L Steven Johnson, Sean R Eddy e Elon Portugaly. Hidden markov model speed heuristic and iterative HMM search procedure. *BMC bioinformatics*, 11(1):431, 2010. Citado na página 2.

[Kan92] Viggo Kann. *On the approximability of NP-complete optimization problems*. Tese de Doutorado, Royal Institute of Technology Stockholm, 1992. Citado na página 22.

[KS06] Arun S Konagurthu e Peter J Stuckey. Optimal sum-of-pairs multiple sequence alignment using incremental carrillo and lipman bounds. *Journal of Computational Biology*, 13(3):668–685, 2006. Citado nas páginas xiii, 20, e 21.

[LBB+07] Mark A Larkin, Gordon Blackshields, NP Brown, R Chenna, Paul A McGettigan, Hamish McWilliam, Franck Valentin, Iain M Wallace, Andreas Wilm, Rodrigo Lopez et al. Clustal W and Clustal X version 2.0. *Bioinformatics*, 23(21):2947–2948, 2007. Citado nas páginas 33 e 34.

[LBZ+00] Harvey F Lodish, Arnold Berk, S Lawrence Zipursky, Paul Matsudaira, David Baltimore, James Darnell et al. *Molecular cell biology*, volume 4. WH Freeman New York, 2000. Citado na página 6.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Em *Soviet physics doklady*, volume 10, páginas 707–710, 1966. Citado na página 10.

[Li03] Kuo-Bin Li. ClustalW-MPI: Clustalw analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586, 2003. Citado nas páginas 3, 35, 50, e 53.

[LR09] Lukasz Ligowski e Witold Rudnicki. An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. Em *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, páginas 1–8. IEEE, 2009. Citado na página 2.

[LSM09a] Yongchao Liu, Bertil Schmidt e Douglas L Maskell. MSA-CUDA: multiple sequence alignment on graphics processing units with CUDA. Em *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, páginas 121–128. IEEE, 2009. Citado nas páginas xiv, 32, 41, 53, e 58.

[LSM09b] Yongchao Liu, Bertil Schmidt e Douglas L Maskell. Parallel reconstruction of neighbor-joining trees for large multiple sequence alignments using CUDA. Em *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, páginas 1–8. IEEE, 2009. Citado nas páginas xiv, xvii, 34, 49, e 54.

[LSVMW07] Weiguo Liu, Bertil Schmidt, Gerrit Voss e Wolfgang Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *Parallel and Distributed Systems, IEEE Transactions on*, 18(9):1270–1281, 2007. Citado nas páginas xiv, 36, 37, e 58.

[MM88] Eugene W Myers e Webb Miller. Optimal alignments in linear space. *Computer applications in the biosciences: CABIOS*, 4(1):11–17, 1988. Citado na página 32.

[Mou04] David W Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2, 2004. Citado na página 2.

[NH96] Cédric Notredame e Desmond G Higgins. SAGA: sequence alignment by genetic algorithm. *Nucleic acids research*, 24(8):1515–1524, 1996. Citado na página 2.

[NHH00] Cédric Notredame, Desmond G Higgins e Jaap Heringa. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of molecular biology*, 302(1):205–217, 2000. Citado nas páginas xiv, 31, 35, e 36.

[NW70] Saul B Needleman e Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. Citado nas páginas 8 e 17.

[Pev05] Jonathan Pevsner. *Bioinformatics and functional genomics*. John Wiley & Sons, 2005. Citado na página 5.

[SKP09] Biswajit Sahoo, Atul Kumar e Sudarsan Padhy. Parallel implementation of centre star method in SMP cluster for multiple sequence alignment. *Int. J. of Recent Trends in Engineering and Technology*, 2(1), 2009. Citado nas páginas xiv, 38, 39, e 58.

[SN87] Naruya Saitou e Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 4(4):406–425, 1987. Citado nas páginas 3, 33, 35, e 48.

[SPD97] Jens Stoye, Sören W Perrey e AWM Dress. Improving the divide-and-conquer approach to sum-of-pairs multiple sequence alignment. *Applied Mathematics Letters*, 10(2):67–73, 1997. Citado na página 38.

[SS73] David Sankoff e Peter H Sellers. Shortcuts, diversions, and maximal chainsin partially ordered sets. *Discrete mathematics*, 4(3):287–293, 1973. Citado na página 9.

[SW81] Temple F Smith e Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981. Citado nas páginas 18 e 37.

[TGP+97] Julie D Thompson, Toby J Gibson, Frédéric Plewniak, François Jeanmougin e Desmond G Higgins. The CLUSTAL_X windows interface: flexible strategies for multiple sequence alignment aided by quality analysis tools. *Nucleic acids research*, 25(24):4876–4882, 1997. Citado na página 34.

[THG94] Julie D Thompson, Desmond G Higgins e Toby J Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673–4680, 1994. Citado nas páginas 2, 31, 33, e 35.

[TKRP05] Julie D Thompson, Patrice Koehl, Raymond Ripp e Olivier Poch. BAliBASE 3.0: latest developments of the multiple sequence alignment benchmark. *Proteins: Structure, Function, and Bioinformatics*, 61(1):127–136, 2005. Citado na página 62.

[TLS+14] Huan Truong, Da Li, Kittisak Sajjapongse, Gavin Conant e Michela Becchi. Large-scale pairwise alignments on GPU clusters: Exploring the implementation space. *Journal of Signal Processing Systems*, 77(1-2):131–149, 2014. Citado nas páginas xiv, 3, 40, 41, e 46.

[Wat86] Michael S Waterman. Multiple sequence alignment by consensus. *Nucleic acids research*, 14(22):9095–9102, 1986. Citado na página 38.

[WJ94] Lusheng Wang e Tao Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology*, 1(4):337–348, 1994. Citado nas páginas 1 e 15.

[Zom06] Albert Y Zomaya. *Handbook of nature-inspired and innovative computing: integrating classical models with emerging technologies*. Springer Science & Business Media, 2006. Citado na página 12.