Alinhamento Global de Várias Sequências Biológicas utilizando Cluster de GPUs

Rodrigo Albuquerque de Oliveira Siqueira

Alinhamento Global de Várias Sequências Biológicas utilizando Cluster de GPUs

Rodrigo Albuquerque de Oliveira Siqueira

Orientador: *Prof. Dr. Marco Aurélio Stefanes*

Monografia entregue a Faculdade de Computação da Universidade Federal de Mato Grosso do Sul - FACOM-UFMS como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

UFMS - Campo Grande/MS
Março/2021

Abstract

A multiple sequence alignment is an important tool for studying and representing similarities between a set of biological sequences – such as *DNAs*, *RNAs* and proteins. This study allows relevant information to be obtained from these sequences, i.e. their functional and evolutionary relations as well as their internal structures. Due to its importance, several methods have been proposed as a solution to this problem. Nonetheless, the problem's inherent complexity, which is described as computationally *NP-Hard*, leads to prohibitive execution times in scenarios with large numbers of lengthy sequences.

In this work, we present a complete implementation of the *Progressive Alignment* heuristic method, using hybrid parallelism for environments with multiple *GPU* devices. This approach allows the construction of global alignments between datasets of numerous lengthy sequences in reasonable time.

Our implementation achieves expressive results, showing *speedups* of up to 380 when compared to the parallel *ClustalW-MPI* aligner for datasets obtained from *NCBI*'s sequence databases.

Resumo

O alinhamento de múltiplas sequências é uma ferramenta importante para o estudo e a representação de similaridades entre conjuntos de sequências biológicas – como *DNAs*, *RNAs* e proteínas. Este estudo permite a obtenção de informações relevantes destas sequências, como suas relações funcionais, evolucionárias e estruturas internas. Devido a sua importância, vários métodos foram propostos como solução a este problema. Entretanto, a complexidade inerente do problema, que é apresentado como computacionalmente *NP-Difícil*, conduz a tempos de execução proibitivos em cenários com muitas sequências longas.

Neste trabalho, apresentamos uma implementação completa para o método heurístico de *Alinhamento Progressivo* utilizando paralelismo híbrido para ambientes com múltiplas *GPUs*. Esta abordagem permite a construção de alinhamentos globais entre bases com muitas sequências de comprimentos longos em tempo razoável.

Nossa implementação atinge resultados expressivos, apresentando *speedups* de até 380 quando comparado ao alinhador paralelo *ClustalW-MPI* para sequências reais obtidas do banco de dados do *NCBI*.

Conteúdo

Lista de Figuras

Lista de Tabelas

CAPÍTULO

1

Introdução

A bioinformática, como um campo científico interdisciplinar, estuda uma miríade de problemas relacionados à biologia com o uso de ferramentas computacionais. No estudo da evolução ou das funções de macromoléculas biológicas – tais como *DNA*, *RNA* ou proteínas –, a necessidade de comparação entre moléculas de diferentes seres ou espécies é um problema comum. Em geral, como as moléculas consideradas são polímeros que podem ser representados por uma sequência de caracteres, compará-las, na prática, consiste em alinhar as sequências correspondentes.

Intuitivamente, um alinhamento busca inserir espaços – ou *gaps* – em cada sequência tal que, dado um esquema para pontuar pares de caracteres e possíveis *gaps*, a pontuação geral do alinhamento seja maximizada ou minimizada. Um alinhamento pode ser global, o qual encontra o melhor alinhamento das sequencias inteiras ou local, que visa encontrar o melhor alinhamento entre certas regiões das sequências. Estes alinhamentos podem ocorrer entre um par ou entre várias sequências. Para um único par de sequências a técnica de programação dinâmica fornece algoritmos de tempo quadrático. Os principais algoritmos desta abordagem são o Algoritmo de Smith-Waterman para a realização de alinhamento local de sequências(S-W) [Smith and Waterman, 1981] e o Algoritmo Needleman–Wunsch para o alinhamento global de duas sequências (N-W) [Needleman and Wunsch, 1970].

O problema de Alinhamento de Várias Sequências – comumente abreviado como *MSA*, do inglês *Multiple Sequence Alignment* – ocupa uma posição de destaque em Biologia Computacional: alinhamentos são utilizados, por exemplo, para inferir possível ancestralidade entre sequências [Mount, 2004], predizer estruturas secundárias de RNA e de proteínas [Ng and Henikoff, 2001], predi-

zer regiões funcionais e auxiliar na construção de árvores filogenéticas.

Soluções ótimas e exatas para o alinhamento de várias sequências são conhecidas e utilizam métodos de programação dinâmica, tal como a estratégia de Carrillo-Lipman [Carrillo and Lipman, 1988]. Entretanto, estas soluções não são usadas na prática devido à sua complexidade. De modo geral, o *MSA* apresenta-se como um problema NP-completo [Wang and Jiang, 1994] e, por isso, soluções exatas exigem um dispêndio considerável de tempo e memória, mesmo para quantidades pequenas de sequências. Ante à inviabilidade de soluções exatas, abandona-se a otimalidade em favor de métodos com complexidades de tempo aceitáveis. Dentre eles se destacam os métodos heurísticos. Estas heurísticas funcionam bem para casos práticos, mantêm a generalidade do problema, são executados em tempo polinomial e fornecem resultados satisfatórios. Dado que o comprimento e a quantidade de sequências podem ser significativamente grandes, o tempo de execução desta heurística pode ser de dias ou semanas, dependendo do caso, mesmo com os métodos mais rápidos.

Existem diversas soluções heurísticas usando diferentes técnicas para o problema de *MSA*. Dentre as quais destacamos *BLAST* [Altschul et al., 1990], *MAFFT* [Katoh et al., 2002], o *Kalign* [Lassmann and Sonnhammer, 2005], o *MUSCLE* [Edgar, 2004], *DeepMSA* [Zhang et al., 2019], a família do *ClustalW* [Thompson et al., 1994], entre outras. Em geral, podemos dividir as abordagens para o MSA em Alinhamento Progressivo e Alinhamento Iterativo. O *Alinhamento Progressivo*, desenvolvido por Da-Fei Feng e Russell Doolittle [Feng and Doolittle, 1987], é uma das abordagens mais amplamente utilizadas principalmente considerando o alinhamento global e a qualidade do alinhamento. De forma resumida, o *Alinhamento Progressivo* requer três etapas sequênciais para a busca de uma solução:

1. *Alinhamento Par-a-Par*: esta etapa consiste do cálculo da pontuação de alinhamento de todas as possíveis permutações de pares entre as sequências, resultando na construção de uma matriz triângular cujas células representam o quanto duas sequências são parecidas, ou seja, a *similaridade* entre as duas sequências.

2. *Construção de Árvore-Guia*: a partir da matriz de similaridade obtida, esta etapa objetiva a construção iterativa de uma estrutura hierárquica, uma árvore binária análoga a uma *árvore filogenética*, cuja função é guiar quais e em que ordem as sequências devem ser alinhadas.

3. *Alinhamento Progressivo*: de acordo com a topologia da *árvore-guia* previamente calculada, a terceira e última etapa executa o alinhamento entre as sequências.

Esta heurística produz uma degeneração no alinhamento final devido à propagação de escolhas ruins tomadas nos passos iniciais do processo. Uma outra abordagem, conhecida como *Alinhamento Iterativo*, permite o realinhamento de sequências, buscando minimizar o impacto causado por erros iniciais no alinhamento, reduzindo a possibilidade de propagá-los para o resultado final. Dentre as soluções iterativas, algumas implementações utilizam os modelos ocultos de Markov [Johnson et al., 2010], e algoritmos genéticos [Notredame and Higgins, 1996]. Em geral, métodos iterativos demandam mais tempo de processamento que métodos progressivos, contudo, são capazes de efetuar buscas mais amplas no espaço de soluções.

Além das heurísticas, outra maneira para a redução do tempo de processamento de uma solução é a utilização de paralelismo, que permite a execução de um algoritmo em vários processadores de forma simultânea. Estes algoritmos podem ser comparados de acordo com sua granulosidade, arquitetura alvo e modelo de paralelização utilizado. Soluções com granulosidade grossa são desenvolvidas para arquiteturas com memória distribuída, como *clusters*, enquanto que soluções com granulosidade fina focam em processadores com múltiplos núcleos e aceleradores, como *FPGAs*, *GPUs* e multi-processadores simétricos. Podemos citar alguns resultados recentes desta abordagem como a versão paralela do MAFFT [Katoh and Toh, 2010a], ferramenta para aceleração do HAlign adotando a plataforma Hadoop [Zou et al., 2015], M2Align, o qual explora cluster de CPU multi-core [Zambrano-Vega et al., 2017], paralelização do ClustalW para sequências grandes [Lopes et al., 2012], além da implementação do primeiro estágio do alinhamento progressivo baseado no ClustalW em um cluster Xeon Phi [Lan et al., 2016].

Há, também, a possibilidade de soluções com granulosidade híbrida, que são desenvolvidas para *clusters* de *GPUs*, utilizando-se de métodos de granulosidade grossa para a comunicação entre *GPUs*. Recentemente várias implementações paralelas híbridas do MSA foram propostas [Chen et al., 2017, Zou et al., 2019, Alawneh et al., 2020, Araujo et al., 2017]. Algumas tratam somente do primeiro estágio (alinhamento par-a-par em Multi-GPU) e outros implementam os três estágios do método alinhamento progressivo. Adicionalmente, existem implementações baseadas no método Neighbor-Joining [Saitou and Nei, 1987a] em uma única GPU para a construção da árvore-guia. Por fim, destacamos a implementação em cluster de CUDA-GPU do alinhamento progressivo de Truong et al [Truong et al., 2014].

Dentre as soluções já existentes, a família Clustal destaca-se na literatura devido à grande variedade de implementações para solucionar o problema de Alinhamento Múltiplo de Sequências. Desta família, são notáveis as contribuições apresentadas pelo ClustalW-MPI [Li, 2003], que implementa paraleli-

zação em *MPI* para a primeira e terceira etapas da heurística de Alinhamento Progressivo, obtendo speedups significativos em relação à versão sequencial do ClustalW. Em sua versão conhecida como ClustalΩ [Sievers et al., 2011], resultados ainda mais expressivos são obtidos através da paralelização de heurísticas distintas. Em razão destes resultados já publicados na literatura, as versões ClustalW-MPI e ClustalΩ serão utilizadas como base de comparação para este trabalho.

Não há na literatura qualquer trabalho em que todas as etapas da heurística de Alinhamento Progressivo estejam implementadas em arquitetura híbrida simultaneamente. Este trabalho propõe uma implementação híbrida completa para o preenchimento desta falta na literatura.

1.1 Objetivos Gerais

O objetivo geral desta dissertação é apresentar uma solução paralela completa, de granulosidade híbrida, para todas as etapas da estratégia de Alinhamento Progressivo para o alinhamento global de várias sequências utilizando um *cluster* de *GPUs* com arquitetura *CUDA*. Nossa solução apresentou resultados expressivos, atingindo *speedups* de até 256 vezes na soma de todas as etapas da heurística, quando comparado com o *ClustalW-MPI* [Li, 2003], uma ferramenta referência em alinhamentos de sequências biológicas, e bastante citada na literatura. Em relação ao *ClustalΩ*, obtemos speedups significativos com todos os conjuntos de dados quando somadas todas as etapas.

1.2 Objetivos Específicos

Entre os objetivos específicos deste trabalho encontram-se a implementação de um *framework* para o fácil desenvolvimento e teste de algoritmos e heurísticas para a solução de problemas relacionados ao Alinhamento de Múltiplas Sequências em ambientes de paralelismo híbrido. Para tal, implementamos os algoritmos para a solução do problema supracitado no *framework* desenvolvido neste trabalho.

Cada algoritmo implementado em paralelismo híbrido obtém também *speedups* significativos em comparação às versões já implementadas na literatura, como exposto nesta dissertação.

1.3 Síntese de Contribuições

O código para a implementação apresentada neste trabalho é público, sob a licença de código aberto *GNU General Public License v3.0*, e está disponível em

https://github.com/rodriados/museqa. Além dos algoritmos apresentados, o repositório implementa um *framework* para futuros trabalhos baseados em paralelismo híbrido com *MPI* e *CUDA*.

1.4 Organização

Este trabalho está dividido da seguinte forma: o Capítulo 2 apresenta os trabalhos relacionados à esta dissertação; o Capítulo 3 exibe definições e conceitos biológicos preliminares; o Capítulo 4 faz uma apresentação do problema do Alinhamento de Várias Sequências Biológicas e das principais abordagens para solucionar o problema; o Capítulo 5 descreve os modelos de paralelismo utilizados; o Capítulo 6 descreve a implementação paralela dos algoritmos; o Capítulo 7 apresenta os experimentos realizados, incluindo os resultados obtidos e análises; e por fim, o Capítulo 8 apresenta a conclusão deste trabalho.

CAPÍTULO

2

Trabalhos Relacionados

A heurística de Alinhamento Progressivo é bastante conhecida e estudada. Além desta abordagem, há vários outros trabalhos com diferentes técnicas na literatura. Nesta seção discutiremos sobre os principais trabalhos publicados anteriormente e descreveremos brevemente sobre suas técnicas e funcionamentos.

Família BLAST

O *BLAST* [Altschul et al., 1990], acrônimo para *Basic Local Alignment Search Tool*, é uma ferramenta para a comparação e busca de sequências biológicas que possibilita aos pesquisadores identificarem partes comuns em uma determinada base de sequências biológicas. Para isso, o *BLAST* provê meios de buscas através da realização de alinhamentos locais, possibilitando a busca e comparação de regiões de interesse nas sequências pesquisadas.

Diferentes versões do *BLAST* foram implementadas, entre elas há versões paralelas implementadas com a utilização de *GPUs* e sistemas de memória distribuída. Destas, destacam-se o *HCudaBLAST* [Khare et al., 2007], que implementa o *BLAST* em um cluster de *GPUs* utilizando o framework *Hadoop* para comunicação entre nós; e o *G-BLASTn* [Zhao and Chu, 2014] que implementa uma versão especializada para proteínas do *BLAST* em uma única *GPU*.

Ainda que extremamente importante para o estudo de sequências biológicas, o *BLAST* não estende sua funcionalidade ao escopo de Alinhamento de Múltiplas Sequências, por não permitir a realização de alinhamentos globais entre sequências.

Família Clustal

O *Clustal* [Higgins and Sharp, 1988] é uma família de *softwares* amplamente utilizada para o problema de *MSA*. Entre suas variações, existem versões com suporte a interface gráfica, como o *ClustalX* [Thompson et al., 1997], e outras com possibilidade de execução paralela em *clusters*. Destaca-se a terceira geração, também conhecida como *ClustalW* [Thompson et al., 1994], responsável por implementar novas funcionalidades e grandes melhorias em relação às gerações anteriores.

Entre as variadas versões do *Clustal*, existem também implementações paralelas, destacando-se o *ClustalW-MPI* [Li, 2003], que consiste de uma ferramenta com os recursos do *Clustal* otimizados para sistemas de computação paralela distribuída, como clusters. Nesta categoria, também há o *MSA-CUDA* [Liu et al., 2009], uma implementação paralela de todas as três etapas do *ClustalW* para sistemas compartilhados, utilizando *GPUs* e atingindo *speedups* significativos em comparação ao *ClustalW-MPI* para uma variedade de grandes *datasets* de sequências de proteínas.

Entretanto, o *MSA-CUDA* se limita à utilização de uma única *GPU*. O uso de várias *GPUs* em um *cluster* possibilita uma redução ainda mais relevante ao tempo de execução do alinhamento.

Uma outra versão importante da família Clustal é a ferramenta *ClustalΩ* [Sievers et al., 2011]. Esta ferramenta pode virtualmente alinhar dezenas de milhares de sequências de proteínas em um tempo razoável e ainda tem a vantagem de entregar alinhamentos de alta qualidade. Ela utiliza a bilioteca OpenMP para realizar a computação dos alinhamentos par-a-par usando multithread. A acurácia em casos de teste pequenos é similar àqueles alinhadores de alta qualidade. Em grandes conjuntos de dados, ClustalΩ supera outras ferramentas em tempo de execução e qualidade. Por estas razões, esta versão é utilizada como uma das bases de comparação para a implementação proposta.

Família Kalign

O *Kalign* [Lassmann and Sonnhammer, 2005] é uma ferramenta para a solução do alinhamento múltiplo global de sequências biológicas. Apesar de também basear-se na heurística de alinhamento progressivo tradicional, o *Kalign* propõe a utilização de algoritmos diferentes para algumas de suas etapas. Em sua primeira versão apresentou-se como uma ferramenta significativamente mais rápida que versões não-paralelas do *ClustalW*, considerando-se execuções em máquinas similares. Além disso, o método mostrou-se mais preciso para conjuntos maiores de sequências e significativamente mais rá-

pido em relação aos métodos iterativos conhecidos.

Em sua terceira versão [Lassmann, 2019], o *Kalign* introduz a implementação de algoritmos paralelos de granulosidade fina, utilizando o *OpenMP*, para estimar a distância entre as sequências e acelerar a contrução de árvores-guias. Com isso, o *Kalign* obtém uma melhoria de performance de até duas ordens de magnitude mais rápido em comparação com soluções não paralelas.

Família MUSCLE

O *MUSCLE* [Edgar, 2004], acrônimo para *Multiple Sequence Comparison by Log-Expectation*, é um software para a resolução do *MSA*, e propõe a heurística *Draft Alignment*, do inglês Alinhamento de Esboço. Esta heurística busca gerar alinhamentos rápidos, mas inicialmente ruins, e melhorá-los em etapas consecutivas de refinamento.

O *MUSCLE* é frequentemente utilizado como um substituto para o *Clustal* e, dependendo das configurações utilizadas, pode gerar alinhamentos melhores que o *Clustal*. Por outro lado, ele é difícil de paralelizar. Apesar disso, MUSCLE-SMP [Deng et al., 2006] foi uma primeira tentativa de paralelização do MUSCLE em sistemas de memória compartilhada e produziu bom desempenho em conjuntos de 50 a 150 proteinas com tamanho médio de 330.

Família T-Coffee

O *T-Coffee* [Notredame et al., 2000], acrônimo para *Tree-based Consistency Objective Function for Alignment Evaluation*, também é um software para o problema de *MSA*, que utiliza a heurística de Alinhamento Progressivo. Uma de suas grandes vantagens é a possibilidade de utilização de diversos formatos de arquivos diferentes, o que permite utilizá-lo para realizar comparações entre os resultados obtidos por softwares diferentes.

Existem diferentes variações do *T-Coffee* disponíveis com propósitos diferentes, como o *R-Coffee* [Wilm et al., 2008] para alinhamentos especializados de *RNA*, e o *TM-Coffee* [Chang et al., 2012] para o alinhamento de proteínas transmembranares. Também existem implementações paralelas para sistemas de memória distribuída [Zola et al., 2007], utilizando-se *MPI*.

Família MAFFT

O MAFFT [Katoh et al., 2002] é uma outra familía bem conhecida para resolver o problema do MSA. Ele introduz duas técnicas novas que reduzem o tempo de execução em CPU. Ele rapidamente identifica regiões homólogas através do uso das tranformadas rápidas de Fourier – FFT. Em sua atualização descrita em [Katoh and Toh, 2007] ela melhora a acurácia e apresenta estas

duas técnicas. O algoritmo PartTree melhora a escalabidade do alinhamento progressivo e a função objetivo de consistência de quatro vias (Four-way consistency) melhora a acurácia dos alinhamentos de ncRNA.

Em [Katoh and Toh, 2010b], todos os estágios do MAFFT foram paralelizados usando a biblioteca POSIX Threads. Com esta abordagem foi obtido um speedup de 10 vezes com diferentes sequências randômicas em um PC com 16 núcleos.

Outros

Existem trabalhos com implementações de algumas etapas do Alinhamento Progressivo paralelizadas com granularidade híbrida, como o trabalho de [Ferlete and Stefanes, 2015, Truong et al., 2014]. Entretanto, ainda não há na literatura qualquer trabalho em que todas as etapas desta estratégia estão implementadas em arquitetura híbrida simultaneamente.

O *DeepMSA* [Zhang et al., 2019] é um método desenvolvido para a construção de alinhamentos múltiplos sensíveis, criado a partir de bases de dados de metagenomas e genomas completos de múltiplas origens, utilizando Modelos Ocultos de Markov. O *DeepMSA* permite a realização de pesquisas eficientes em múltiplas bases de dados com sequências grandes e biologicamente distantes.

O *Pfam* [Sonnhammer et al., 1998] consiste de um banco de dados de famílias completas de proteínas com anotações de alinhamentos múltiplos gerados utilizando-se Modelos Ocultos de Markov (HMM, do inglês *Hidden Markov Models*), afim de complementar análises de proteínas com o *BLAST*. Ao fornecer domínios completos de famílias de proteínas, o *Pfam* facilita a rotulação de sequências biologicamente significativas e, em alguns casos, detecções mais sensíveis.

Ainda que o *Pfam* não forneça ferramentas ou proponha algoritmos para alinhamentos locais ou globais de sequências, as suas bases de dados permitem que pesquisadores realizem estudos mais aprofundados de novas sequências, ao disponibilizar acesso a um grande número de alinhamentos múltiplos rotulados.

CAPÍTULO

3

Conceitos Preliminares

Este capítulo apresenta definições necessárias para a compreensão e desenvolvimento do trabalho proposto, e uma breve descrição dos conceitos biológicos e computacionais. Também estabelece a linguagem utilizada e fixa notações.

3.1 Conceitos Biológicos

Todo organismo vivo possui macromoléculas responsáveis por coordenar o seu desenvolvimento e funcionamento. Estas moléculas caracterizam-se por serem polímeros constituídos de longas cadeias de bases nitrogenadas (no caso do *DNA* e *RNA*), ou aminoácidos (no caso de proteínas). As bases nitrogenadas são 5 diferentes compostos[a], que se organizam em pares, emparelhando-se com suas respectivas bases complementares. Já os aminoácidos somam 20 compostos, como listados na Tabela 3.1.

No contexto da biologia, o alinhamento de sequências é relevante para o entendimento das relações entre sequências biológicas. Caso duas sequências biológicas compartilhem um ancestral comum, pequenos *mismatches* – ou seja, alinhamentos incorretos – podem ser interpretados como mutações pontuais, e espaços vazios – os *gaps* – como mutações de inserção ou remoção introduzidas em uma ou ambas sequências desde sua divergência inicial no tempo.

No alinhamento de sequências proteicas, a similaridade entre aminoácidos ocupantes de uma certa posição particular na sequência pode ser in-

[a]Para formar o *DNA*, as bases utilizadas são: Adenina (A), Citosina (C), Guanina (G), e Timina (T). Para o *RNA*, substitui-se a Timina por Uracila (U).

Aminoácido	Símbolo	Abreviação
Glicina ou Glicocola	Gly, Gli	G
Alanina	Ala	A
Leucina	Leu	L
Valina	Val	V
Isoceulina	Ile	I
Prolina	Pro	P
Fenilalanina	Phe, 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 3.1: Listagem de Aminoácidos.

terpretada como uma medida de quão *conservada* uma região se encontra entre as diferentes linhagens, o que indica que a presença de tais regiões foram preservadas pela seleção natural [M. Cooper and Brown, 2008], sugerindo a existência de importância funcional ou estrutural à região conservada [Ng and Henikoff, 2001]. Uma sequência altamente conservada é aquela que permaneceu relativamente inalterada entre seres cujos ancestrais comuns encontram-se em nós longínquos da *árvore filogenética*.

Uma árvore filogenética – também conhecida como árvore evolutiva – é uma representação gráfica em forma de árvore que apresenta as relações evolutivas entre diversas espécies biológicas ou outras entidades. Nesta árvore, cada nó é conhecido como uma *unidade taxonômica* e representa uma espécie ou um ser (diretamente observado através das sequências ou hipotetisado a partir de um alinhamento), e cada aresta representa uma relação de ancestralidade entre dois nós. Comumente, o comprimento de uma aresta também representa a diferença de tempo geológico entre as entidades.

Neste trabalho, o termo *unidade taxonômica operacional*, ou simplesmente *UTO*, será empregado para referir-se às sequências ou aos seus respectivos alinhamentos.

3.2 Alfabetos, Símbolos e Sequências

Um alfabeto Σ é um conjunto finito e não-vazio. Um elemento $\sigma \in \Sigma$ é chamado *caractere* ou *símbolo*. De forma geral, uma *sequência* s é uma sucessão ordenada sobre o alfabeto Σ tal que $s = (\sigma_1, \sigma_2, \ldots, \sigma_l) \in \Sigma^l$, onde $l \geq 0$ é um inteiro. Por conveniência e simplicidade, denotaremos s por $\sigma_1\sigma_2\ldots\sigma_l$ apenas. Se $s = \sigma_1\sigma_2\ldots\sigma_l$, definimos o *comprimento* $|s|$ de s por $|s| = l$. Se $l = 0$, e portanto $|s| = 0$, então s é chamada *sequência vazia* e é denotada por ε.

Um *segmento* $s[i\ldots j]$ de uma sequência s, para $i \leq j$, é a sequência definida por $s[i\ldots j] = \sigma_i\sigma_{i+1}\ldots\sigma_j$. Se $i > j$, definimos $s[i\ldots j] = \varepsilon$, ou seja, uma sequência vazia. Por comodidade, denotaremos o caso particular em que $j = i$ por $s[i]$ simplesmente. Nesse caso, claramente $s[i] = \sigma_i$. Usaremos esta notação frequentemente em descrições de algoritmos para fazer referência aos símbolos de s.

Se Σ é um alfabeto tal que $\textvisiblespace \notin \Sigma$, denotamos por Σ' o alfabeto $\Sigma' = \Sigma \cup \{\textvisiblespace\}$. O símbolo $\textvisiblespace$ é tratado como especial e é denominado *espaço vazio* ou *gap*.

Em nosso texto, supomos que Σ é o alfabeto correspondente às bases nitrogenadas presentes em sequências de *DNA* e *RNA*, ou aos aminoácidos presentes em sequências proteicas.

3.3 Alinhamento de Sequências

Sejam $k \geq 2$ um inteiro positivo e $s_1, s_2, \ldots, s_k$ sequências sobre um alfabeto Σ, com $|s_i| = l_i$ para $i = 1, 2, \ldots, k$. Um *alinhamento* A de $s_1, s_2, \ldots, s_k$ é uma matriz $A = A(A_{ij})$ de dimensões $k \times l$ com entradas de $\Sigma' = \Sigma \cup \{\textvisiblespace\}$ (e $l \geq max_{i=1}^{k}\{l_i\}$) tal que a linha A_i do alinhamento A consista exatamente da sequência s_i com possíveis *gaps* inseridos entre os caracteres de s_i.

Dizemos 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. Um alinhamento mostra quais partes de cada sequência podem ser comparadas a outras, sugerindo assim como tranformar uma sequência em outra através da substituição, inserção ou remoção de símbolos. Um alinhamento pode ser visualizado colocando cada sequência uma embaixo da outra como mostrado na figura sequinte com dois diferentes alinhamentos das sequências $(abacb, bacb, aacc)$. A parte esquerda representa o alinhamento $abacb, \textvisiblespace bacb, a\textvisiblespace acc)$ e o lado direito representa o alinhamento $(aba\textvisiblespace cb\textvisiblespace, \textvisiblespace b\textvisiblespace acb\textvisiblespace, a\textvisiblespace a\textvisiblespace c\textvisiblespace c)$. Note que o primento sugere que o último c na terceira sequência vem da operação de substituição e o segundo alinhamento sugere que ele vem da operação de inserção. Diversos critérios podem ser utilizados em uma avaliação qualitativa entre os dois alinhamentos, como o comprimento final do alinhamento obtido. Utilizando-se este critério como

exemplo, podemos assumir que o primeiro alinhamento é qualitativamente melhor que o segundo.

$$\begin{array}{ccccc} a & b & a & c & b \\ \textvisiblespace & b & a & c & b \\ a & \textvisiblespace & a & c & c \end{array} \qquad \begin{array}{ccccccc} a & b & a & \textvisiblespace & c & b & \textvisiblespace \\ \textvisiblespace & b & \textvisiblespace & a & c & b & \textvisiblespace \\ a & \textvisiblespace & a & \textvisiblespace & c & \textvisiblespace & c \end{array}$$

Seja $A = (A_{ij})$ um alinhamento de $s_1, s_2, \ldots, s_k$. Um espaço de s_i no alinhamento A é qualquer segmento não-vazio maximal de A_i, composto apenas de espaços (*gaps*), isto é, um espaço de s_i em A é um segmento $A_i[j \ldots j']$ de A_i tal que: $j \leq j'$, todos os símbolos de $A_i[j \ldots j']$ são $\textvisiblespace$, ou não é possível estender $A_i[j \ldots j']$ em A_i por meio de *gaps*, ou seja, $j = 1$ ou $A_i[j-1] \neq \textvisiblespace$ e $j' = n$ ou $A_i[j+1] \neq \textvisiblespace$.

3.4 Tipos de Alinhamentos

Há pelo menos três maneiras distintas de se alinhar duas sequências. Os chamados *alinhamentos globais*, foco exclusivo desta dissertação, procuram comparar as sequências de entrada 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 alinhar genes or proteínas homólogas.

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

De forma semelhante, pode-se estar interessado em comparar sequências que podem não ser muito parecidas globalmente, mas que possuam trechos altamente semelhantes (não necessáriamente prefixos ou sufixos). Este alinhamento é conhecido como *alinhamento local*, sendo desejável para identificar se há trechos altamente conservados entre as duas sequências.

3.5 Distância de Edição e Similaridade

Para efetuarmos comparação entre sequências precisamos estabelecer certos critérios, como uma medida de comparação. Uma maneira é a contagem mínima de operações de inclusão, exclusão ou substituições de caracteres para transformar uma sequência na outra. Este critério é uma medida bem conhecida, chamada de *distância de edição* ou *distância de Levenshtein* [Levenshtein, 1966]. A distância de edição, sendo um métrica, apre-

senta propriedades de reflexibilidade, não-negatividade, positividade, comutatividade e desigualdade triangular. Essas propriedades permitem a garantia de otimalidade de uma das soluções encontradas.

Uma outra opção é comparar as sequências de modo, a saber, o quão iguais elas são. Neste caso, busca-se o cálculo da máxima *similaridade* entre as sequências. Para isto, utiliza-se uma *matriz de substituição* que indica a pontuação para alinhar dois símbolos diferentes e a penalização para a adição de *gaps* no alinhamento. Pela falta de rigor em relação às propriedades de métrica da matriz de substituição, e ao fato de buscar-se o máximo valor, não há garantias de que o cálculo de similaridade propicie a otimalidade de uma solução. Ainda assim, devido às propriedades biológicas trazidas aos alinhamentos com o uso de matrizes de substituição, medidas de similaridade serão utilizadas neste trabalho devido.

3.6 Matrizes de Pontuação e Substituição

Ao realizar o calculo de similaridade entre duas sequências s_1 e s_2, é necessário atribuir uma determinada pontuação para o alinhamento dos símbolos presentes nas sequências. Esta pontuação é geralmente obtida através de uma matriz, conhecida como *matriz de pontuação*.

Seja $\Sigma' = \Sigma \cup \{\textvisiblespace\}$, uma matriz de pontuação é uma matriz que possui números reais como entradas e são indexadas nas linhas e colunas pelos elementos de Σ'. Para $a, b \in \Sigma'$ 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 (ou a pontuação) para a operação de substituição se $a, b \in \Sigma$, e de inserção se $a = \textvisiblespace$, e de remoção se $b = \textvisiblespace$. A entrada $\gamma_{\textvisiblespace \to \textvisiblespace}$ não é definida. As operações de substituição, inserção e remoção sobre uma sequência são chamadas de *operações de edição*.

Utilizando a matriz de pontuação γ, define-se a pontuação ou custo $c_\gamma(A)$ do alinhamento A entre s_1 e s_2 como $c_\gamma(A) = \sum_{i=1}^{l} \gamma_{s_1[i] \to s_2[i]}$. Definimos também que $c(s_1, \ldots, s_k)$ é o custo do alinhamento ótimo entre as sequências $s_1, \ldots, s_k$.

A Figura 3.1 mostra um exemplo de matriz de pontuação, utilizado para encontrar distâncias entre duas sequências. Nesta matriz, o alinhamento de símbolos iguais obtém pontuação 1. Já para o alinhamento de símbolos diferentes, inserção ou remoção de símbolos, a pontuação obtida é 0.

$\gamma =$

	a	b	␣
a	1	0	0
b	0	1	0
␣	0	0	

Figura 3.1: Exemplo de matriz de pontuação.

No contexto do alinhamento de sequências de proteínas, a similaridade é

calculada através do uso de uma *matriz de substituição*, com a pontuação para todas as possíveis trocas de um aminoácido por outro. Entre as matrizes mais relevantes, encontram-se versões de *PAM* (do inglês: *Percent Accepted Matrix*) [Dayhoff et al., 1978] e *BLOSUM* (do inglês: *Blocks of Amino Acid Substitution Matrix*) [Henikoff, 1992].

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

Os valores de uma matriz *BLOSUM* são calculados conforme a Equação 3.1, 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 se encontrar os aminoácidos i e j de forma aleatória em qualquer sequência de proteína. O fator λ é um fator escalar, definido de tal forma que a matriz contenha apenas valores inteiros [Zomaya, 2006].

$$B_{ij} = \left(\frac{1}{\lambda}\right) \log\left(\frac{p_{ij}}{f_i \times f_j}\right) \tag{3.1}$$

3.7 Problema do Alinhamento de Várias Sequências

De forma intuitiva, um alinhamento entre k sequências $s_1, \ldots, s_k$ é obtido inserindo-se *gaps* nas sequências e então colocando-as uma sobre a outra de forma que cada caractere ou *gap* esteja emparelhado a um único caractere ou *gap* de cada sequência. O uso de programação dinâmica para alinhamentos globais iniciou-se com Needleman-Wunsch [Needleman and Wunsch, 1970] para encontrar o alinhamento ótimo entre duas sequências, maximizando a pontuação entre duas sequências com um número mínimo de operações de inserção e exclusão, sendo conhecido como critério de máxima similaridade. Algoritmos para alinhamento de um par de sequências baseados em programação dinâmica geralmente possuem complexidade de tempo da ordem de $O(l^2)$, onde l corresponde ao tamanho das sequências a serem alinhadas. Na Figura 3.2 ilustramos um alinhamento de várias sequências.

Formalmente podemos definir o problema do Alinhamento de Várias Sequências como:

Problema 3.1 (Alinhamento de Várias Sequências) *Dados um inteiro $k \geq 2$ e k sequências $s_1, \ldots, s_k$ sobre um alfabeto Σ e fixada uma função de pontuação $\gamma = \Sigma' \times \Sigma' \to \mathbb{R}_{\geq 0}$, encontrar um alinhamento A cujo custo $c_\gamma(A)$ seja $c(s_1, \ldots, s_k)$.*

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

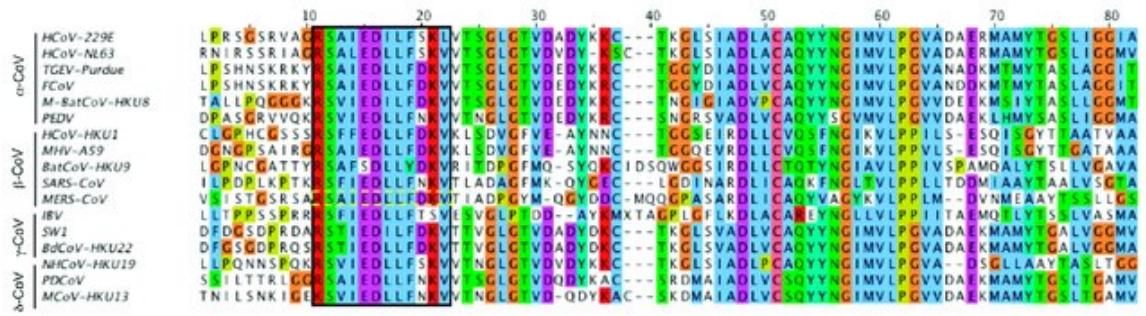

Figura 3.2: Alinhamento de 17 sequências parciais da proteína Spike do Coronavírus representando a similaridade entre elas. Nesta figura, *gaps* são representados por hífens [Alsaadi et al., 2019].

Muitos métodos diferentes são propostos para resolução deste problema, tais como: métodos progressivos [Feng and Doolittle, 1987], programação dinâmica [Carrillo and Lipman, 1988], alinhamento de soma de pares, alinhamento de consenso, dentre outros [Chan et al., 1992]. O *MSA* possui características de otimização combinatória, e por ser um problema NP-difícil [Wang and Jiang, 1994] uma solução exata se torna inviável para quantidades grandes de sequências, tornando o uso de heurísticas uma das possíveis abordagens para este problema.

Estes métodos possuem complexidade de tempo variando de $O(l^k)$, para métodos baseados em programação dinâmica, até $O(l^2k)$, para métodos heurísticos como o Alinhamento Progressivo, onde k é o número de sequências e l o tamanho médio das sequências [Chan et al., 1992].

Ainda que a utilização de programação dinâmica seja uma maneira direta de se resolver o *MSA*, o custo desta abordagem é proibitivo em termos de tempo e memória. Heurísticas como o *Alinhamento Progressivo* têm sido soluções viáveis utilizadas para superar estas restrições. Devido ao grande número de sequências e de símbolos em cada uma delas (na ordem de milhares), o *Alinhamento Progressivo*, mesmo com uma redução significativa da complexidade computacional, ainda demanda muito tempo para encontrar uma solução. Para reduzirmos ainda mais este consumo, utilizaremos arquiteturas paralelas neste trabalho.

3.7.1 Método de Carrillo-Lipman

No contexto de soluções para o problema de alinhamento de duas ou mais sequências, encontra-se também o método proposto por Carrillo e Lipman [Carrillo and Lipman, 1988]. A ideia é determinar um caminho $\mu = (s_1, \ldots, s_k)$ entre as sequências $s_1, \ldots, s_k$, onde k é o número de sequências, limitando o espaço de busca de soluções e reduzindo significativamente a quantidade de cálculos.

Seja $M(\mu)$ uma medida de similaridade entre as sequências $s_1,\ldots,s_n$ que representa o score de cada caminho μ, existe no mínimo um caminho ótimo $\mu* = (s_1,\ldots,s_k)$ cuja similiridade é máxima para cada $M(\mu)$.

Considere um reticulado k-dimensional $L = (s_1,\ldots,s_k)$. Cada vértice em L pode ter o canto final de um sub-reticulado $L = (s_1[i_1],\ldots,s_k[i_k])$, onde para cada sequência s, $s[i]$ denota o segmento de sequência composto dos primeiros i elementos de s.

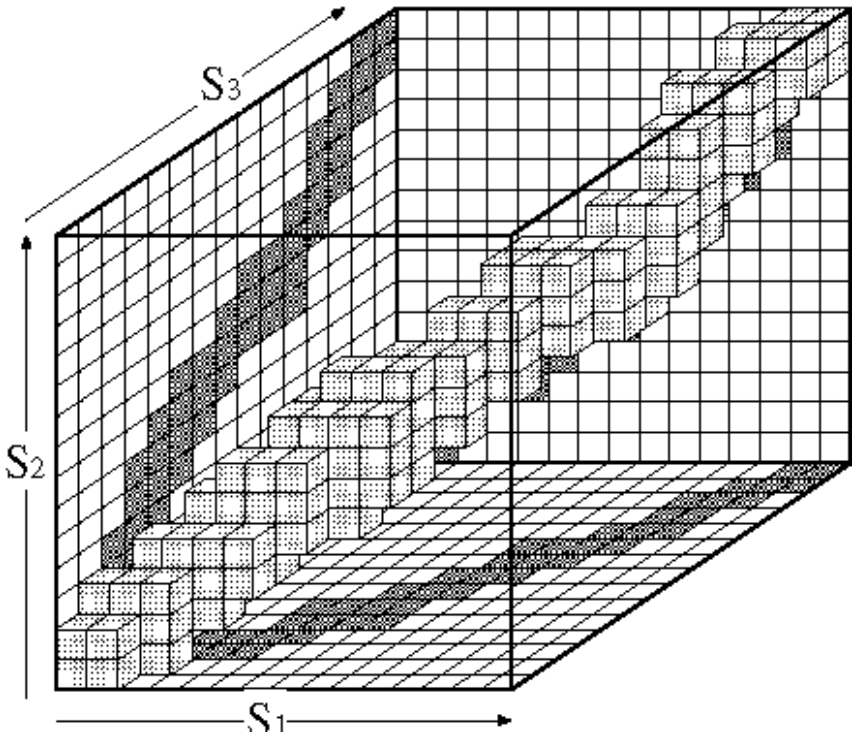

Figura 3.3: O método de Carrillo-Lipman divide o espaço de busca em sub-reticulados para a construção de um alinhamento global de múltiplas sequências. Nesta ilustração, apenas três sequências são alinhadas. Fonte: [Masuno et al., 2007].

O método consiste em encontrar, de forma recursiva, o caminho ótimo para todo sub-reticulado de L. Computa-se inicialmente a pontuação de cada possível caminho μ^L no cubo que possui posição o vértice no canto inicial. No próximo passo é calculada a similaridade máxima para alcançar o canto inicial para os vértices adjacentes do cubo através de caminho válido. Esse processo, ilustrado pela Figura 3.3, é repetido até que o canto final do cubo seja alcançado, de forma que o caminho tenha similaridade máxima μ^*. É possível calcular μ^* para as sequências $(s_1,\ldots,s_k)$ de tamanho $(l_1,\ldots,l_k)$ em $O(\sum_{i<j}^{k} l_i l_j)$ passos.

Normalmente o alinhamento de várias sequências implica a realização de um alinhamento entre $\binom{k}{2}$ pares de sequências. Ao observar μ^* é necessário considerar somente os caminhos μ em $L = (s_1, s_k)$ que satisfaçam a medida de similaridade entre as sequências s_i e s_j, chamado de caminho X_{lk}. Portanto, o

caminho μ no conjunto $X = \bigcap_{i<j}^{k} X_{ij}$ são todos os possíveis caminhos candidatos a serem caminhos ótimos.

Desta maneira, é possível informar um limiar mínimo para $M(\mu^*)$, possivelmente encontrado por um método heurístico mais rápido, para que podas possam ser realizadas em sub-reticulados que produzirão caminhos μ^L, tal que $max(M(\mu^L)) < M(\mu^*)$. O alinhamento ótimo A encontrado pelo método de Carrillo-Lipman pode, então, ser interpretado como um "refinamento"da solução A obtida por algum outro método de alinhamento de várias sequências.

3.7.2 Modelos Ocultos de Markov

Um *Modelo Oculto de Markov*, ou simplesmente *HMM*, acrônimo para *Hidden Markov Model*, é um modelo estatístico que pode ser usado para descrever a evolução de eventos observáveis dependentes de determinados fatores internos que não são diretamente observáveis [Yoon, 2009]. Um *HMM* consiste de dois processos estocásticos, a saber, um processo invisível de estados ocultos e um processo visível de eventos observados. Os estados ocultos formam uma *Cadeia de Markov*, e a distribuição probabilística dos eventos observados é dependente do estado subjacente.

A modelagem de observações utilizando-se estas duas camadas, uma visível e outra invisível, mostra-se bastante útil já que muitos problemas reais lidam com a classificação de observações brutas em um determinado número de categorias ou rótulos que nos são mais significativos. Apesar de ser frequentemente utilizada para reconhecimento de voz, esta abordagem tem se mostrado também bastante efetiva para a modelagem de sequências biológicas, tais como sequências de *DNA* e proteínas.

Para solução do problema de *MSA*, entretanto, um *HMM* precisa ser treinado com grandes quantidades de sequências previamente alinhadas para obter-se um bom resultado. Desta forma, métodos heurísticos mais rápidos como o Alinhamento Progressivo tornam-se úteis para auxiliar com a geração de alinhamentos para treinamento destes modelos.

3.8 O Método de Alinhamento Progressivo

Como visto na Seção 3.7.1, métodos que utilizam o paradigma de programação dinâmica, como o proposto por Carrillo-Lipman não são eficientes para um grande número de sequências ou com sequências muito longas. Como alternativa a este paradigma, temos heurísticas para alinhar várias sequências, como o Alinhamento Progressivo proposto por Da-Fei Feng e Russell Doolittle [Feng and Doolittle, 1987], que produz o alinhamento de k sequências a partir de vários alinhamentos par-a-par para resolução do problema de MSA.

O Alinhamento Progressivo é, geralmente, composto de três etapas, sendo que os métodos utilizados por cada etapa podem variar de acordo com o algoritmo utilizando esta heurística.

Sequencialmente, as três etapas da heurística de Alinhamento Progessivo são: cálculo da matriz de distâncias entre os alinhamentos par-a-par de todas sequências, construção de árvore-guia e alinhamento progressivo. A Figura 3.4 ilustra esquematicamente as três etapas de um Alinhamento Progressivo.

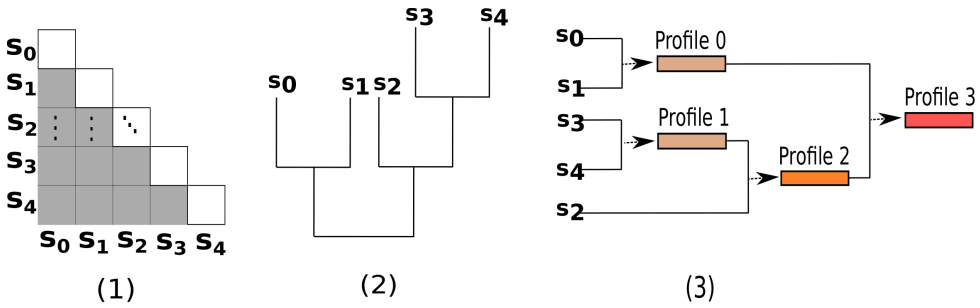

Figura 3.4: As três etapas da heurística de alinhamento progressivo: (1) alinhamento par-a-par (PW), (2) árvore-guia (NJ), (3) alinhamento progressivo (PA). A árvore-guia possui raiz e os *Profiles* são múltiplas sequências.

Esta heurística é utilizada por trabalhos na literatura como o *T-Coffee* [Notredame et al., 2000], o *MUSCLE* [Edgar, 2004] e várias versões da família *Clustal* [Thompson et al., 1994]. Destes, um dos mais populares para se resolver o Alinhamento de Múltiplas Sequências é o algoritmo utilizando pelo *ClustalW* que, como um programa que utiliza a heurística de Alinhamento Progressivo, adiciona sequências uma a uma a um alinhamento existente, criando-se assim um novo alinhamento. A ordem pela qual as sequências são adicionadas ao alinhamento é definida por uma árvore-guia, que é computada utilizando o custo do alinhamento entre os possíveis alinhamentos dos pares de sequências.

Para k sequências de tamanho l, temos que a primeira etapa do algoritmo tem complexidade de tempo $O(k^2l^2)$ para o cálculo da matriz de distâncias. Na segunda etapa, a construção da árvore-guia é concluída com complexidade de tempo $O(k^4)$. Por sua vez, a terceira e última etapa tem complexidade $O(k^3 + kl^2)$ para a construção do alinhamento de múltiplas sequências. Com isso, o algoritmo tem complexidade total de $O(k^4 + l^2)$.

A seguir detalhamos cada uma das etapas do Método Alinhamento Progressivo.

3.8.1 Matriz de Distâncias

Em sua primeira etapa, a heurística de Alinhamento Progressivo busca construir uma matriz de distâncias entre todos os pares de sequências a se-

rem alinhadas. Isto é, para a solução desta etapa, requere-se a execução de um algoritmo de alinhamento global sobre todas as possíveis combinações de pares de um conjunto de k sequências, resultando-se num total de $k^2 - k$ distâncias calculadas, que constituem a matriz de distâncias resultante desta etapa.

Dentre os algoritmos utilizados para a solução desta etapa, destacam-se algoritmos de programação dinâmica como o proposto por Needleman e Wunsch [Needleman and Wunsch, 1970]. A ideia por trás deste algoritmo é o cálculo da pontuação de todos os alinhamentos entre todos os prefixos de um par de sequências, de forma que o melhor alinhamento global seja obtido pelo resultado do alinhamento dos prefixos que igualam as sequências originais em sua totalidade. Ainda que a proposta de Needleman-Wunsch permita construir o alinhamento global de máxima pontuação entre duas sequências, apenas a pontuação deste alinhamento é relevante para esta etapa.

3.8.2 Árvore-Guia

A partir das distâncias entre as sequências, constrói-se uma árvore-guia ou árvore filogenética. Nesta etapa da heurística, determinamos a ordem dos alinhamentos, que serão passados à próxima etapa, com o intuito de encontrar uma árvore que relacione as sequências em uma ordem evolucionária.

Um algoritmo para a solução desta etapa é a proposta de Sokal e Michenel, conhecida como *UPGMA* [Sokal and Michener, 1958], acrônimo para *Unweighted Pair Group with Arithmetic Mean*. A ideia deste algoritmo é realizar um agrupamento hierárquico aglomerativo simples sobre um conjunto de dados descrito por uma matriz de distância entre seus elementos.

Apesar de ser comumente utilizado em diversas áreas, como inteligência artificial e análises estatísticas, o uso de *UPGMA* é criticado em algumas áreas da bioinformática, como a filogenética, por não inferir bem relacionamentos entre sequências de proteínas sem que haja uma justificativa específica para o seu uso em um conjunto de dados a ser analisado.

A fim de ser uma alternativa mais relevante à bioinformática, Saitou e Nei propuseram o algoritmo *Neighbor-Joining* (*NJ*) [Saitou and Nei, 1987b]. Este algoritmo realiza uma adaptação ao *UPGMA*, para que a distância entre cada par de UTOs seja considerada e recalculada a cada iteração de agrupamento de UTOs durante a construção da árvore-guia. O aumento de qualidade da árvore gerada, porém, traz consigo um significativo aumento de complexidade de tempo para esta etapa da heurística.

Após o trabalho original de Saitou e Nei, foram publicadas propostas para atenuar a complexidade imposta pelo *Neighbor-Joining*, tais como *RapidNJ* [Simonsen et al., 2008] e *NINJA* [Wheeler, 2009], que utilizam-se de algumas

estruturas de dados para evitar recalcular dados que podem ser reutilizados para iterações seguintes do algoritmo original.

Dentre as alternativas para a solução desta etapa, encontram-se também métodos estatísticos a exemplo da *Estimativa por Máxima Verossimilhança* [Felsenstein, 1981] ou *Inferência Bayesiana*, utilizados por ferramentas como o *FastTree* [Price et al., 2010] e o *PhyML* [Guindon and Gascuel, 2003]. Apesar de estes métodos geralmente produzirem árvores-guias de maior qualidade, de acordo com [Ogden and Rosenberg, 2006], várias ferramentas como o *ClustalW* [Higgins and Sharp, 1988] e o *MSA-CUDA* [Liu et al., 2009] optam pelos métodos mais simples, baseados no *UPGMA*, devido à complexidade de tempo significativamente menor.

3.8.3 Alinhamento Progressivo

Por fim, a etapa de Alinhamento Progressivo, que dá nome ao método heurístico como um todo, é responsável pela construção final do alinhamento global entre todas as sequências a partir da árvora-guia construída na etapa anterior.

O algoritmo de Needleman-Wunsch [Needleman and Wunsch, 1970], adaptado para o cálculo de distâncias de alinhamentos par-a-par na etapa 1, apresenta-se como uma solução também a etapa de alinhamento progressivo. Para isto, adaptações ao algoritmo original são necessárias para permitir o alinhamento de UTOs compostos por mais de uma única sequência.

Uma outra alternativa para esta etapa é o método de Myers-Miller [Myers and Miller, 1988], cuja proposta é semelhante àquela de Needleman-Wunsch porém com mudanças que promovem maior eficiência de uso de memória ao reconstruir um alinhamento entre cada par de sequências ou perfis. Em suma, Myers e Miller propuseram a execução do algoritmo de Needleman-Wunsch em duas frentes, dividindo-se o espaço de busca de soluções para um alinhamento encontrando-se o ponto central ótimo de um subalinhamento recursivamente, até que o alinhamento global seja obtido.

CAPÍTULO

4

Alinhamento Progressivo de Várias Sequências

Como descrito na Seção 3.8, a heurística de Alinhamento Progressivo consiste de três etapas que são executadas em sequência. A primeira etapa realiza a contrução de uma matriz de distâncias entre as sequências, calculando a pontuação de todas as combinações de alinhamentos par-a-par entre as sequências. A segunda etapa constrói uma árvore-guia a partir da matriz de distâncias da primeira etapa. Por fim, na terceira etapa constrói alinhamentos progressivos a partir da topologia da árvore-guia.

Este capítulo apresenta os algoritmos sequenciais utilizados como base para este trabalho em cada uma destas etapas da heurística de *Alinhamento Progressivo* para a execução do *MSA*.

4.1 Etapa 1: Alinhamento Par-a-par

Para a primeira etapa da heurística, para o alinhamento par-a-par, utilizamos o algoritmo de Needleman-Wunsch (*NW*) [Needleman and Wunsch, 1970]. Este método foi uma das primeiras aplicações de programação dinâmica para a comparação de sequências biológicas e essencialmente divide um problema em uma série de pequenos problemas, e utiliza as soluções de problemas menores para encontrar a solução ótima para o problema maior. O *NW* realiza o alinhamento global ótimo entre um par de sequências s_1 e s_2, com $|s_1| = l_1$, $|s_2| = l_2$, em que l_1 e l_2 são comprimentos das sequências s_1 e s_2 respectivamente. É realizado o cálculo de uma matriz bidimensional D, onde o elemento $D_{i,j}$ representa a pontuação do melhor alinhamento entre os prefixos $s_1[1 \ldots i]$ e

$s_2[1 \ldots j]$. Para calcular esta matriz é utilizada programação dinâmica. Para o início do processamento, a primeira linha e a primeira coluna da matriz D são preenchidas com a penalidade de *gap* $g \times i$, sendo g a penalidade de *gap* e i o tamanho da sequência não-vazia terminando no i-ésimo elemento. As outras coordenadas serão calculadas por meio da fórmula de recorrência descrita pela Equação 4.1.

$$D_{i,j} = max \begin{cases} D_{i-1,j-1} + \gamma_{s_1[i] \to s_2[j]} & \text{Alinhar símbolos iguais ou diferentes} \\ D_{i-1,j} + g & \text{Símbolo } gap \text{ na sequência } s_1 \\ D_{i,j-1} + g & \text{Símbolo } gap \text{ na sequência } s_2 \end{cases} \tag{4.1}$$

Na Equação 4.1, $\gamma_{s_1[i] \to s_2[j]}$ é o custo para alinhar o elemento $s_1[i]$ com $s_2[j]$ de acordo com a matriz de substituição γ, e g é o valor da comparação de qualquer elemento com um *gap*.

A equação é aplicada para preencher a matriz D, iniciando-se do canto superior esquerdo até o canto inferior direito, com complexidade de tempo $O(l_1 \times l_2)$. Para se encontrar o alinhamento, mantém-se um ponteiro em cada célula da matriz indicando qual célula vizinha originou o valor, possibilitando percorrer estes ponteiros no sentido inverso. Este passo é conhecido como *traceback*. Em alguns casos, o *traceback* poderá percorrer mais de um possível caminho, permitindo a geração de mais de um alinhamento global ótimo.

Para a conclusão desta etapa, este método é aplicado para alinhar todos os $\binom{k}{2}$ possíveis pares de k sequências, produzindo uma matriz triangular de similaridades entre as sequências.

Algoritmo 1: Algoritmo de *Needleman-Wunsch*.

Input: Sequências s_1 e s_2, matriz de substituição γ, penalidade de *gap* g
Output: Similaridade entre s_1 e s_2

```
1  Seja D_{i,j} a matriz com a pontuação ótima do alinhamento entre os
   prefixos s1[1...i] e s2[1...j].
2  for i ← 0 to |s1| do
3      D_{i,0} = g × i;
4  for j ← 0 to |s2| do
5      D_{0,j} = g × j;
6  for i ← 1 to |s1| do
7      for j ← 1 to |s2| do
8          done = D_{i-1,j-1} + γ_{s1[i]→s2[j]};
9          top = D_{i-1,j} + g;
10         left = D_{i,j-1} + g;
11         D_{i,j} = max(done, top, left);
12 return D_{|s1|,|s2|}
```

4.2 Etapa 2: Construção de Árvore-Guia

A partir da matriz de similaridade (ou também matriz de distância) D obtida, a segunda etapa realiza a construção de uma *árvore-guia*, análoga a uma árvore filogenética, que determinará a ordem dos alinhamentos na última etapa, com a ideia de tentar hierarquizar as sequências de uma forma evolucionária.

Devido à maior qualidade apresentada em relação ao UPGMA, conforme discutido na Seção 3.8, para esta etapa utilizamos o método Neighbor-Joining (*NJ*) [Saitou and Nei, 1987b], cujo objetivo é construir uma árvore e obter o tamanho dos ramos desta árvore juntando-se as sequências, que representam, cada uma, *unidades taxonômicas operacionais* (*UTOs*) vizinhas. Um par de *UTOs* são vizinhos quando compartilham um único nó interior em uma árvore bifurcada e sem raiz.

O método *NJ* inicia com uma árvore estrela, tal que todas as *UTOs* sejam vizinhas entre si. O método realiza junções sucessivas em *UTOs* vizinhas gerando uma nova *UTO* correspondente ao alinhamento entre as *UTOs* agrupadas. Este processo é repetido até que restem apenas 3 *UTOs* vizinhas.

O primeiro passo do método consiste em identificar o par de *UTOs* tal que a árvore tenha a menor soma de ramos. Tal operação pode ser determinada aplicando a Equação 4.2 em todos os pares de *UTOs*:

$$Q(i,j) = (k-2)D_{i,j} - \left(\sum_{n=1}^{k} D_{i,k} + \sum_{n=1}^{k} D_{j,k} \right) \tag{4.2}$$

onde k é a quantidade de sequências. Com os valores de i e j tais que Q seja mínimo, podemos adicionar uma nova *UTO* u, que é composta por i e j, com os novos valores para a matriz de similaridade. A Equação 4.3 é utilizada para calcular novos valores da matriz de similaridade entre u e as demais *UTOs* que não fazem parte de u. As Equações 4.4 e 4.5 são utilizadas para o cálculo da distância entre i e u, e entre j e u respectivamente [Saitou and Nei, 1987b]:

$$D^{l}_{u,n} = \frac{1}{2}\left(D^{l-1}_{i,n} + D^{l-1}_{j,n} - D^{l-1}_{i,j} \right), \text{ para } n \neq i,j \tag{4.3}$$

$$\delta(i,u) = \frac{1}{2} D^{l-1}_{i,j} + \frac{1}{2(k-2)} \left(\sum_{n=1}^{k} D^{l-1}_{i,n} - \sum_{n=1}^{k} D^{l-1}_{j,n} \right) \tag{4.4}$$

$$\delta(j,u) = D^{l-1}_{i,j} - \delta(i,u) \tag{4.5}$$

Este método possui complexidade de tempo $O(k^3)$ e gera uma árvore sem raiz com ramos proporcionais à divergência estimada entre cada ramo. A raiz da árvore é posicionada onde ambos os lados da árvore tenham o mesmo

tamanho.

A Figura 4.1 ilustra sucintamente o funcionamento do método a partir da matriz de distâncias abaixo.

	a	b	c	d	e
a	0	5	9	9	8
b	5	0	10	10	9
c	9	10	0	8	7
d	9	10	8	0	3
e	8	9	7	3	0

Tabela 4.1: Matriz de distâncias usada como entrada do método NJ que resultará na árvore ilustrada na Figura 4.1

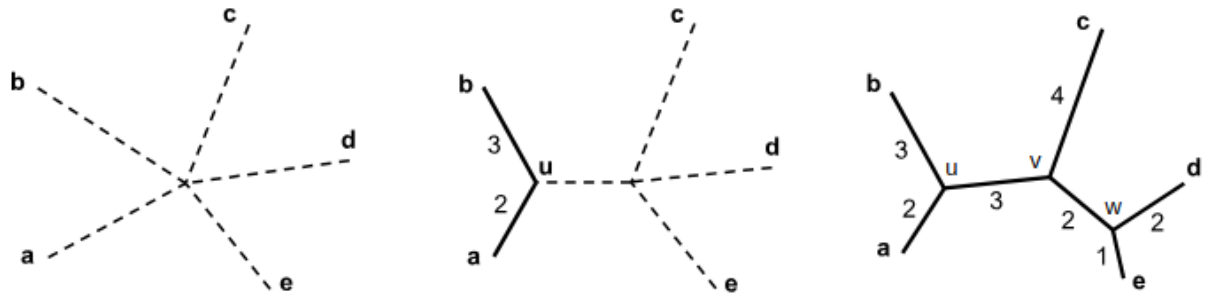

Figura 4.1: Execução do Método Neighbor-Joining com as 5 sequências (a,b,c,d,e) da matriz anterior. Neste caso, apenas duas iterações são suficientes para construir uma árvore-guia completa: a junção de a e b, seguida pela junção de d e e. Os ramos da árvore-guia estão etiquetadas com suas respectivas distâncias.

4.3 Etapa 3: Alinhamento Progressivo

Uma solução para o alinhamento progressivo de várias sequências é o algoritmo *JUNTA* [de Brito, 2003]. A ideia básica deste método é "colar" dois ou mais alinhamentos para obter um alinhamento maior composto por todas as sequências dos alinhamentos menores, de acordo com a topologia da *árvore-guia* obtida.

Este método funciona tomando alinhamentos $A_1,\ldots,A_r$ em que exatamente uma das sequências de entrada s seja comum a cada par de alinhamentos e esteja na primeira linha de cada alinhamento. Utilizando 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*.

Seja $s = s[1],\ldots,s[l]$ uma sequência de comprimento l, e o alinhamento A_i possua k_i sequências e comprimento n_i, para $i = 1,\ldots,r$. Fixado i, seja $p_{i,j}$ a

coluna do j-ésimo símbolo de s no alinhamento A_i para todo $j = 1,\ldots,l$, define-se $p_{i,0} = 0$ e $p_{i,l+1} = n_i + 1$. Seja $z_{i,j} = p_{i,j} - p_{i,j-1} - 1$, para todo $j = 1,\ldots,l+1$. A interpretação para $z_{i,j}$ de acordo com a definição é que $z_{i,j}$ é o número 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 $j = 1,\ldots,l+1$, ou seja, o maior número de espaços que ocorre imediatamente antes de $s[j]$ em qualquer dos alinhamentos de A_i.

Um fato interessante do *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 final. Isso é condizente com as palavras clássicas de Feng e Doolittle [Feng and Doolittle, 1987]: "uma vez um *gap*, sempre um *gap*".

Ainda que o *JUNTA* seja mais rápido por realizar alinhamentos apenas observando-se uma sequência em cada UTO, os algoritmos baseados no método de Needleman-Wunsch apresentam-se como alternativas de maior qualidade. Este aumento de qualidade, entretanto, é obtido em troca de maior uso de memória para reconstruir cada alinhamento entre UTOs.

O método por Myers e Miller [Myers and Miller, 1988] apresenta-se como uma alternativa eficiente para a solução desta etapa. Este algoritmo propõe uma evolução ao método de Needleman-Wusch, reduzindo-se a complexidade de espaço em memória necessário para a reconstrução do alinhamento final entre duas UTOs.

A proposta de Myers-Miller é encontrar um "ponto central"de uma conversão ótima de s_1 em s_2 utilizando-se uma execução "normal" e outra "invertida" do algoritmo de Needleman-Wunsch para o cálculo de pontuação apenas. Então, uma conversão ótima pode ser encontrada determinando-se recursivamente conversões ótimas de ambos lados deste ponto central.

Seja $\rho^i(s) = s[1\ldots i]$ um prefixo contendo os i primeiros símbolos de uma UTO s, e sejam s_1 e s_2 UTOs a serem alinhadas com comprimentos $|s_1| = l_1 > 1$ e $|s_2| = l_2 > 0$, respectivamente. Seja $i^* = \left\lfloor \frac{l_1}{2} \right\rfloor$, tal que i^* divida a matrix D em duas partes de tamanhos iguais. Na fase normal, aplica-se o método para cálculo de pontuação às UTOs $\rho^{i^*}(s_1)$ e s_2, resultando em dois vetores que satisfazem:

$CC(j) =$ pontuação para a conversão de $\rho^{i^*}(s_1)$ em $\rho^j(s_2)$
$DD(j) =$ pontuação para a conversão de $\rho^{i^*}(s_1)$ em $\rho^j(s_2)$ com remoção ao fim

Seja $s^T = (\sigma_l, \sigma_{l-1}, \ldots, \sigma_1)$ uma sequência invertida de s onde $l = |s|$, e seja $\tau^i(s) = (\sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_l)$ um sufixo de s. Podemos notar que $\rho^{l-i}(s^T)^T = \tau^i$. Na fase invertida, aplica-se o método para cálculo de pontuação às UTOs $\rho^{l_1-i^*}(s_1^T)$

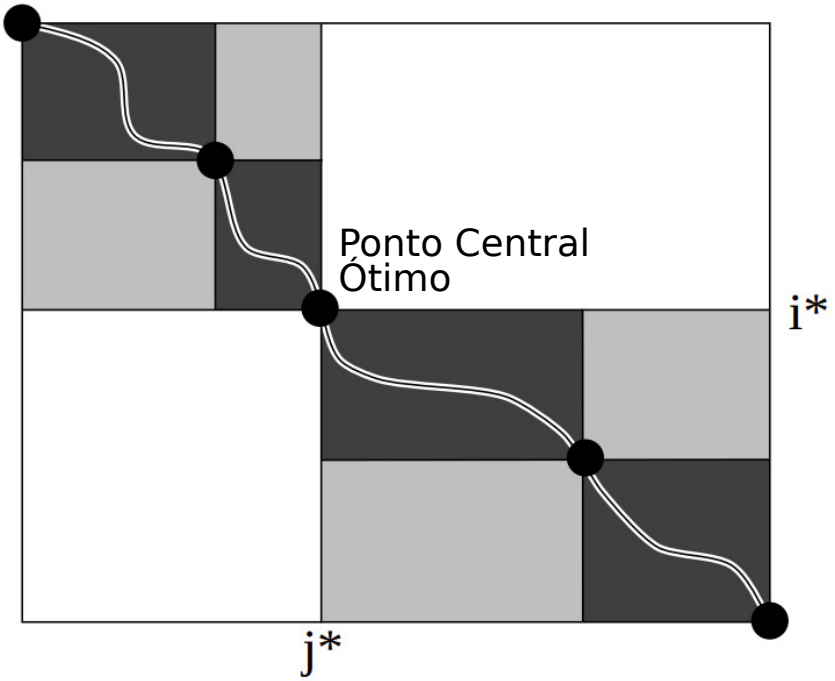

Figura 4.2: Um ponto central divide o problema em sub-problemas que são resolvidos recursivamente [Myers and Miller, 1988].

e s_2^T, produzindo-se novos vetores *RR* e *SS* com os papéis de *CC* e *DD*, que satisfazem:

> $RR(l_2 - j) =$ pontuação para a conversão de $\tau^{i^*}(s_1)$ em $\tau^j(s_2)$
> $SS(l_2 - j) =$ pontuação para a conversão de $\tau^{i^*}(s_1)$ em $\tau^j(s_2)$ com remoção no início

Desta forma, a pontuação ótima para a conversão de s_1 em s_2 é

$$\max_{j=0}^{l_2} \left(\max \begin{cases} CC(j) + RR(l_2 - j) \\ DD(j) + SS(l_2 - j) - g \end{cases} \right) \tag{4.6}$$

Se a pontuação máxima é encontrada em j^*, então (i^*, j^*) é ponto central ótimo para o alinhamento. Quando mais de um ponto possa ser ótimo, o critério de desempate pode ser definido livremente pela implementação.

Dado um ponto central ótimo (i^*, j^*), um alinhamento global entre as UTOs s_1 e s_2 pode então ser construído recursivamente através da concatenação do alinhamento entre $\rho^{i^*}(s_1)$ e $\rho^{j^*}(s_2)$, e do alinhamento entre $\tau^{i^*}(s_1)$ e $\tau^{j^*}(s_2)$. A Figura 4.2 ilustra a divisão do alinhamento em sub-problemas a partir de um ponto central [Myers and Miller, 1988].

CAPÍTULO

5

Modelos de Paralelismo

Tradicionalmente, algoritmos e softwares são desenvolvidos para computação sequencial. Um problema é dividido em uma sequência de instruções que devem ser executadas em ordem por um único processador. Como apenas uma instrução pode executar em um determinado momento, soluções sequenciais que requerem grandes números de instruções gastam muito tempo para sua execução.

De maneira simplificada, na computação paralela são utilizados vários recursos computacionais simultaneamente. Para isso, busca-se dividir o problema em partes independentes e que podem ser resolvidas simultaneamente em diferentes processadores, empregando um mecanismo geral de sincronismo ou coordenação das operações.

A memória de um computador paralelo pode ser compartilhada ou distribuída. Ao utilizar memória compartilhada, vários processos podem acessar o mesmo espaço de endereçamento. As *GPUs* aproveitam-se desta arquitetura de memória. Em arquiteturas que utilizam memória fisicamente distribuída, cada memória local só pode ser acessada por seu processador e a sincronização requer comunicação entre processos através de troca de mensagens. O *MPI*, acrônimo para *Message Passing Interface*, é um padrão de comunicação de dados baseado principalmente em arquiteturas de memória distribuída [Gropp et al., 1999]. Existem ainda arquiteturas de memória híbridas que utilizam de memória compartilhada e distribuída.

Processos paralelos podem ser caracterizados através de sua granulosidade, isto é, a razão entre tempo de computação e tempo de comunicação. Na granulosidade grossa, cada processo gasta relativamente grandes quantidades de tempo executando instruções sequenciais. Por outro lado, a granulosidade

fina caracteriza-se quando os processos executam poucas instruções sequenciais entre operações de comunicação.

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

5.1 GPUs

Os computadores mais modernos atualmente estão dotados de dispositivos especializados em processamento de imagens conhecidos como *GPUs*. Estes dispositivos proporcionam um número de operações gráficas para a *CPU*, como renderização de imagens na memória e posterior exibição em tela.

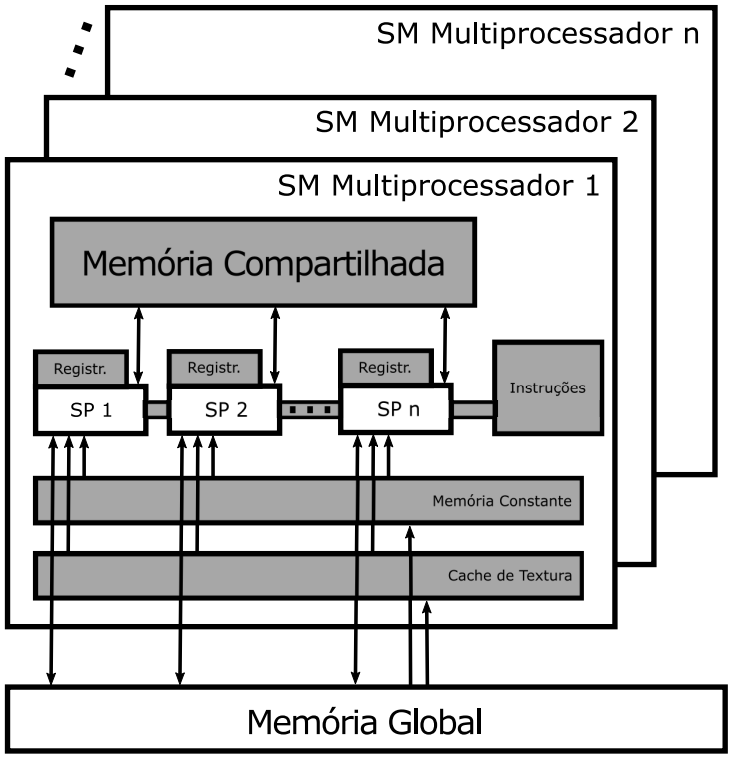

Figura 5.1: Diagrama simplificado da arquitetura CUDA [Cook, 2012].

Devido à natureza intrinsecamente paralela do processamento gráfico, a Nvidia® viu a oportunidade de trazer as *GPUs* para o destaque em computação de alta performance, acrescentando a seus dispositivos a ferramenta de programação *CUDA* - do inglês *Compute Unified Device Architecture*. Um programa em *CUDA* possui a capacidade de executar instruções de programação geral na *GPU* e é composto por uma parte sequencial executada na *CPU* e uma parte paralela executada nos *kernels* da *GPU*. Por sua vez, um *kernel* é composto por um conjunto de *threads* organizadas em *grids*. Cada *grid* é composto por blocos de *threads* que são sincronizadas e podem fazer comunicação entre si através de uma memória compartilhada por bloco. As *threads*

de diferentes blocos podem se comunicar através de acessos à memória global da *GPU*, que é compartilhada por todas as *threads* [Cook, 2012], o que pode ser visto na Figura 5.1.

Uma vez compilado, os *kernels* são executados por *CUDA cores* (anotados como *SM* na Figura 5.1, de *Streaming Multiprocessor*) que consistem de muitas *threads* divididas em *warps*. Cada *warp* contêm 32 *threads* que executam o mesmo código em paralelo. Ainda que as *threads* de um bloco possam executar blocos diferentes de um mesmo programa, a maior eficiência e o melhor desempenho ocorrem quando todas as *threads* seguem o mesmo fluxo de execução de um programa [Cook, 2012].

5.2 MPI

O *MPI* é um padrão de comunicação de dados em computação paralela que disponibiliza especificação de funções que podem ser utilizadas em diversas linguagens de programação, como C/C++, Fortran, Python e Java. O objetivo do *MPI* é possibilitar a criação de programas portáveis que exploram a existência de múltiplos processadores através da troca de mensagens [Gropp et al., 1999].

Na especificação do *MPI*, uma aplicação é constituída por um ou mais processos que se comunicam, utilizando-se de funções para o envio e recebimento de mensagens entre os processsos. Cada processo é identificado por um valor inteiro sequencial, único e não-negativo, ou seja, se há p processos, então cada um será identificado por um valor entre $0, 1, \ldots, p-1$. Além disso, cada processo pode executar um programa diferente, permitindo que múltiplos programas operem sobre múltiplos dados, oferecendo flexibilidade para ambientes de memória compartilhada ou distribuída.

O *MPI* disponibiliza a seus processos mecanismos de comunicação ponto-a-ponto, em que mensagens são trocadas entre apenas dois processos específicos, e operações coletivas em que um grupo de processos executa uma função de forma sincronizada. O *MPI* ainda é capaz de realizar comunicação assíncrona e programação modular, através de *comunicadores* que permite ao programador encapsular estruturas de comunicação interna da aplicação.

5.3 Arquitetura Híbrida

Para uma arquitetura híbrida combinam-se elementos de paralelismo de granularidades fina e grossa. Nesta arquitetura, requerem-se elementos de coordenação e sicronização entre múltiplos dispositivos independentes com capacidade de paralelismo em memória compartilhada.

Define-se como uma arquitetura híbrida homogênea aquela em que todos os nós de trabalho de um *cluster* possuem as mesmas características, configurações e capacidade de processamento. Neste cenário, procura-se uma divisão igualitária de trabalho entre cada nó, assumindo-se que os nós são indistinguíveis em capacidade de trabalho. Por outro lado, em arquiteturas heterogêneas, faz-se necessário considerar as diferenças de configuração entre cada nó para maximar a eficiência de uso da capacidade computacional disponível.

Uma configuração comum para esta arquitetura são clusteres cujos nós comunicam-se utilizando redes ou barramentos, onde a cada nó ou processo é disponibilizado o acesso a um processador *multi-core* ou a uma *GPU*, conforme mostra a Figura 5.2.

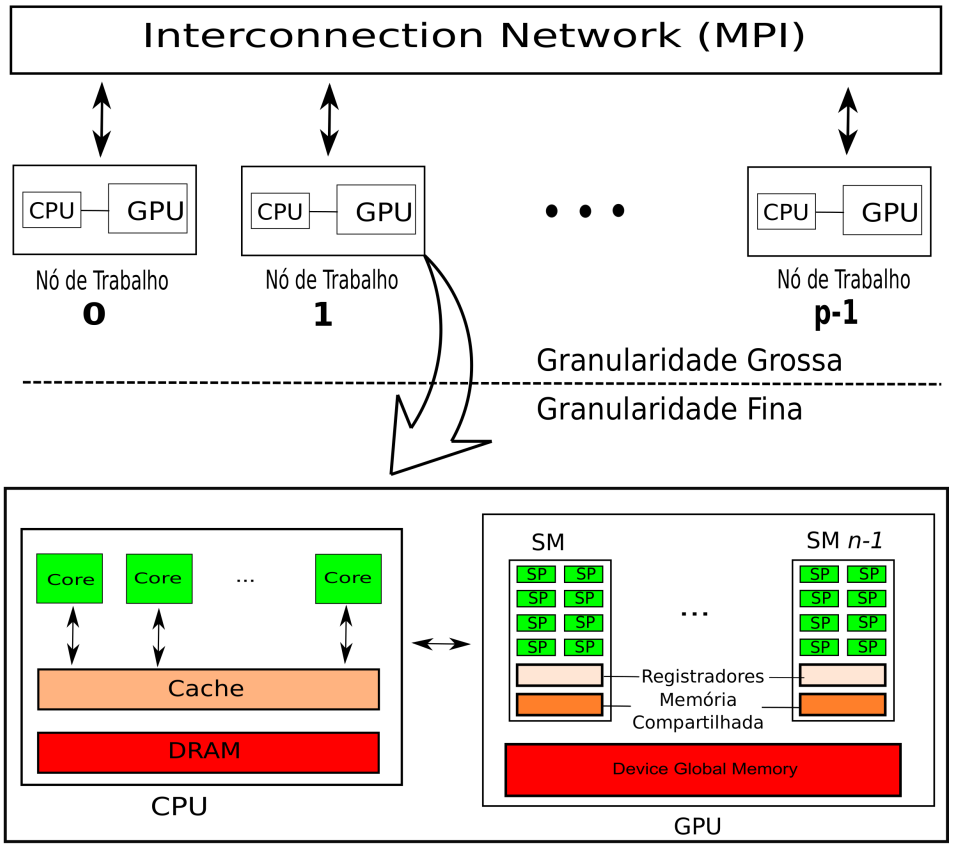

Figura 5.2: Modelo de uma arquitetura híbrida homogênea.

Um problema paralelizável em arquiteturas híbridas deve possuir características que possibilitem sua divisão em sub-problemas para uma arquitetura de memória distribuída. Cada sub-problema, por sua vez, deve possuir características que permitam a utilização de soluções paralelizáveis em arquiteturas de memória compartilhada. Idealmente, soluções híbridas devem requerer poucos pontos de sincronia entre sub-problemas designados a diferentes processadores de memória distribuída.

Neste trabalho, a arquitetura híbrida utilizada consiste de *clusters* homogê-

neos conectados por *MPI*, onde cada nó possui o acesso a uma ou mais *GPUs* capazes de executar programas *CUDA*. Assumiremos sempre que o número de nós de trabalho em um *cluster* é igual ao número de *GPUs* utilizadas no sistema. Ademais, cada nó de trabalho terá acesso exclusivo a uma, e apenas uma *GPU* para realizar seu processamento.

CAPÍTULO

6

Implementação Paralela Híbrida para o Alinhamento Progressivo

Neste trabalho considera-se que um nó é um processo *MPI*. Os nós de trabalho estão direta e exclusivamente relacionados a uma *GPU* de um cluster ou máquina. Sendo assim, uma máquina equipada com mais que uma *GPU* pode hospedar mais que um nó de trabalho, mesmo que possua apenas uma única *CPU*. O nó-mestre, único em cada execução, pode existir em qualquer máquina da rede, mesmo que esta não possua *GPUs* ou já hospede um nó de trabalho.

Neste capítulo serão descritas as propostas de implementação de paralelização do *MSA*, nas três etapas da heurística *Alinhamento Progressivo* em dois níveis de granulosidade. Na granulosidade grossa, a execução de cada etapa é distribuída entre o nó-mestre e os nós de trabalho utilizando *MPI*. Na granulosidade fina, as tarefas de cada nó de trabalho são paralelizadas através do uso de *GPUs* com arquitetura *CUDA*. O resultado dos cálculos da primeira etapa são distribuídos a todos os nós, de forma que a matriz de similaridades seja conhecida por todos os nós antes do início da segunda etapa.

6.1 Distribuição de Sequências entre Processos

Inicialmente, o nó-mestre realiza a leitura de um ou mais arquivos de entrada no padrão *FASTA* contendo um total de k sequências biológicas. Para otimizar o uso de memória, cada sequência é imediatamente compactada obtendo-se economia de até um terço do espaço requerido pela sequência em seu estado original, como esquematizado na Figura 6.1.

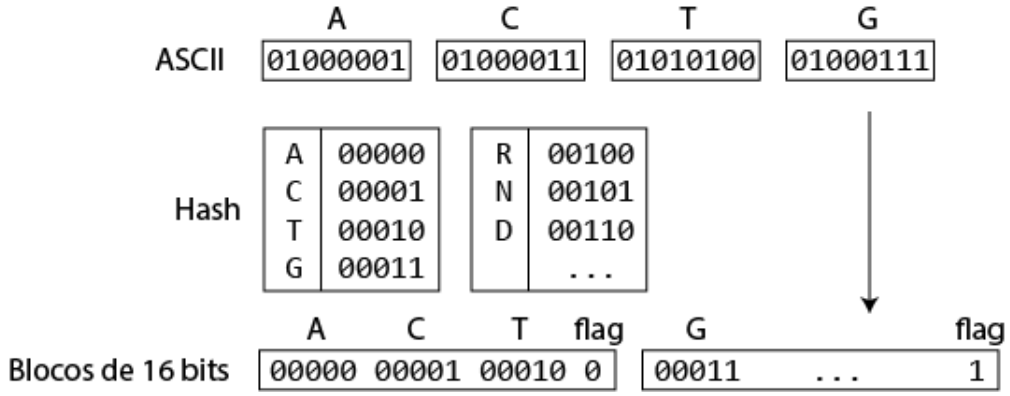

Figura 6.1: Modelo de compactação utilizado para o armazenamento de sequências. Cada caractere da sequência é transformado por uma tabela de hashing, para uma representação em 5 bits. A *flag* indica se o bloco é o último da sequência. Desta forma, em 16 bits armazena-se caracteres que normalmente ocupariam 24 bits. *Gaps*, extensões de *gaps*, fins de sequências e caracteres inválidos também podem ser representados.

Duas alternativas foram estudadas para o envio de sequências aos nós de trabalho. A primeira opção visa designar alternadamente a cada nó os pares a serem por ele alinhados, de forma que nem todo nó de trabalho necessite de todas as sequências. A segunda opção consiste do envio completo de todas as sequências a todos os nós. Devido a otimizações realizadas pelo método de distribuição do *MPI* e a possibilidade de uma versatilidade maior para a posterior distribuição de pares, optou-se pela segunda opção. Assim sendo, todas as k sequências são enviadas para todos os p nós de trabalho, preservando-se a ordem original da leitura das sequências.

6.2 Etapa 1: Alinhamento Par-a-Par Paralelo

Com as sequências já disponíveis em todos os nós, cada nó de trabalho gera uma lista, em ordem lexicográfica, contendo todos os pares de sequências que deve alinhar. Esta lista é gerada de forma autônoma por cada nó a partir de seu identificador no comunicador *MPI*. No total, devem ser alinhados $\binom{k}{2}$ pares para a construção completa da matriz de similaridades, e cada nó é responsável por executar $\binom{k}{2}/p$ alinhamentos.

Para garantir que o trabalho realizado por cada nó seja sempre consistente, independentemente do número de nós e da quantidade de sequências a serem processadas, os pares de sequências são gerados de maneira que sua ordenação global seja sempre a mesma. Seja $i > 0$ o identificador de um nó, e seja $N = \binom{k}{2}$ o total global de pares a serem alinhados. O nó i gerará e será responsável por executar todos os pares dentro do intervalo $\left[(i-1)\frac{N}{p}, i\frac{N}{p} - 1\right]$. Seja $j = (x,y)$ o identificador de um par composto pelas sequências x e y, onde $j \in [0,N]$. A Equação 6.1 garante que um par j sempre alinhe as mesmas

sequências x e y, mesmo que o número total de sequências ou nós de trabalho varie em diferentes configurações de execução.

$$\begin{aligned} x &= \left\lfloor \sqrt{2j+2} + \frac{1}{2} \right\rfloor \\ y &= j - \binom{x-1}{2} \end{aligned} \tag{6.1}$$

Após a geração de seus pares, cada nó trabalha de maneira autônoma e somente reporta ao nó-mestre o resultado de seus cálculos ao fim de sua execução. Para calcular a pontuação do alinhamento de cada par de sequências, cada nó reserva na memória global de sua *GPU* espaço suficiente para armazenar as sequências que deve alinhar, a matriz de substituição escolhida, o resultado final dos alinhamentos e eventuais caches auxiliares necessários, sempre em observância ao total de memória disponível no equipamento no momento. Para permitir que acessos mais rápidos sejam realizados, copia-se a matriz de substituição e uma das sequências do par a uma região de memória compartilhada de cada bloco de *threads* em execução, efetivamente trazendo ganhos cumulativos a cada acesso.

O cálculo da similaridade entre as sequências de cada par é realizado pelo algoritmo *LazyRScan-mNW* [Truong et al., 2014] que utiliza programação dinâmica e a fórmula de recorrência proposta por Needleman e Wunsch [Needleman and Wunsch, 1970], descrito na Seção 4.1. Cada par é processado por um bloco de *threads* na *GPU*, de forma que vários pares são alinhados simultaneamente como ilustrado na Figura 6.2.

O algoritmo *LazyRScan-mNW* é uma proposta de paralelização do método de Needleman-Wunsch, calculando-se a matriz D em uma frente de execução diagonal, de forma que cada *thread* fique responsável por uma linha desta matriz e dependa somente do último valor calculado pela *thread* da linha imediatamente anterior. Os valores calculados por esta frente de execução são armazenados em uma região de memória compartilhada no dispositivo. Esta paralelização causa o efeito de requerer um "tempo de aquecimento"até que todas as *threads* estejam ativas e produzam máximo *throughput*. Ainda que este efeito seja proporcional ao número de *threads* utilizadas em cada bloco, o impacto causado pelo tempo de espera é inversamente proporcional ao tamanho das sequências a serem alinhadas.

Devido à baixa quantidade de memória compartilhada disponível em cada bloco, faz-se necessária a execução do alinhamento em *slices*, em que resultados parciais de um alinhamento são armazenados em memória global. Este resultado é então utlizado para retomar a execução de uma linha da matriz D após a conclusão do cálculo de um *slice* anterior.

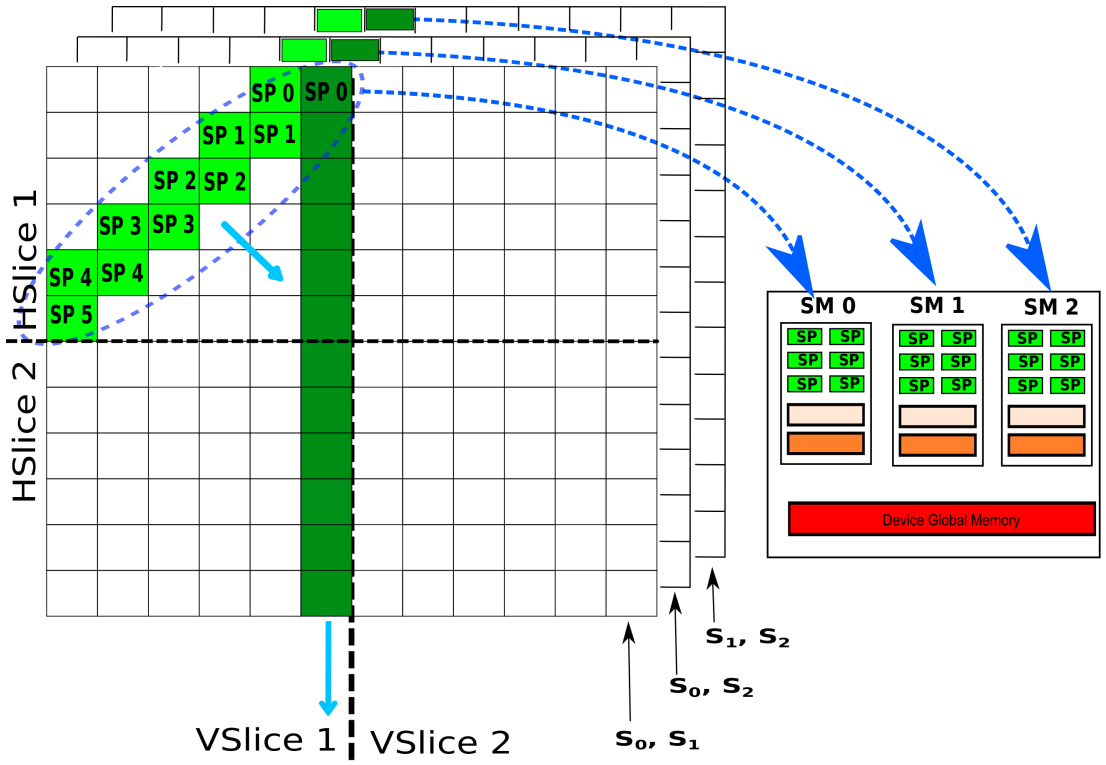

Figura 6.2: Execução do *LazyRScan-mNW* em uma GPU. Cada bloco (*SM*) é responsável pelo alinhamento de um par utilizando-se uma frente de execução diagonal. Múltiplos pares diferentes são alinhados simultaneamente em blocos de *threads* distintos. O alinhamento de um par é realizado em *slices* devido à quantidade limitada de memória compartilhada em um único bloco. Resultados parciais de cada *slice*, em verde escuro. Após uma *thread* (*SP*) atingir o limite da *slice* e armazenar seu resultado parcial em memória global (*vslice* na imagem), seu trabalho é imediatamente redirecionado à próxima linha livre (*hslice*) da matriz D a ser calculada.

O uso da memória global da *GPU* é otimizado, de forma que o espaço alocado para auxiliar a execução seja sobrescrito apenas esporadicamente com resultados parciais dos alinhamentos. Esta otimização permite que cada par requisite apenas $O(l)$ de memória global do dispositivo para alinhar suas sequências, sendo l o comprimento da maior sequência do par. Cada nó de trabalho tentará executar o maior número de pares simultaneamente, sendo limitado apenas pela disponibilidade da memória global de sua *GPU*. Caso não seja possível acomodar todos os pares do nó simultaneamente, a tarefa será realizada em vários passos, utilizando-se da mesma estratégia.

Após o término da execução de todos os pares de um nó, os resultados são copiados da memória da *GPU* para a memória do nó, mantendo-se a ordem estritamente igual àquela de seus pares gerados anteriormente. Assim que a execução por todos os nós é finalizada, seus resultados são distribuídos a todos os outros nós, incluindo o nó-mestre, de forma que, ao fim desta primeira etapa da heurística, todos os nós possuam a pontuação dos alinhamentos de todos os $\binom{k}{2}$ pares de sequências.

Ainda que a presença das similaridades de todos os alinhamentos permita a construção de uma matriz triangular real, optou-se por manter estes resultados em uma lista ordenada para que a etapa seguinte possua liberdade para o uso de diferentes estruturas de dados.

6.3 Etapa 2: Construção de Árvore-Guia

Para a construção da árvore-guia, utilizamos o algoritmo *Neighbor-Joining* (*NJ*) [Saitou and Nei, 1987b]. Ainda que já tenha sido implementado para a utilização em uma única *GPU* [Liu et al., 2009] [Zheng et al., 2012], o *NJ* apresenta um desafio para sua paralelização em granulosidade híbrida devido à grande dependência de dados durante sua execução.

A etapa de construção de uma árvore-guia inicia-se com a transformação da lista de similaridades entre pares de sequências, obtida da etapa anterior, em uma matriz D simétrica com k linhas e k colunas. Cada linha e coluna em D representa uma *UTO* conectada ao nó raiz de uma árvore estrela, como descrito na Seção 4.2. Devido às sucessivas transformações que a matriz D sofrerá durante cada iteração para a execução desta etapa, construímos também uma lista de índices que relacionam uma *UTO* conectada à árvore estrela, a linhas e colunas ativas em D, que se justifica para a redução da complexidade de calculo e acesso à matriz D^{l+1} a partir de D^l em uma iteração l posterior do algoritmo.

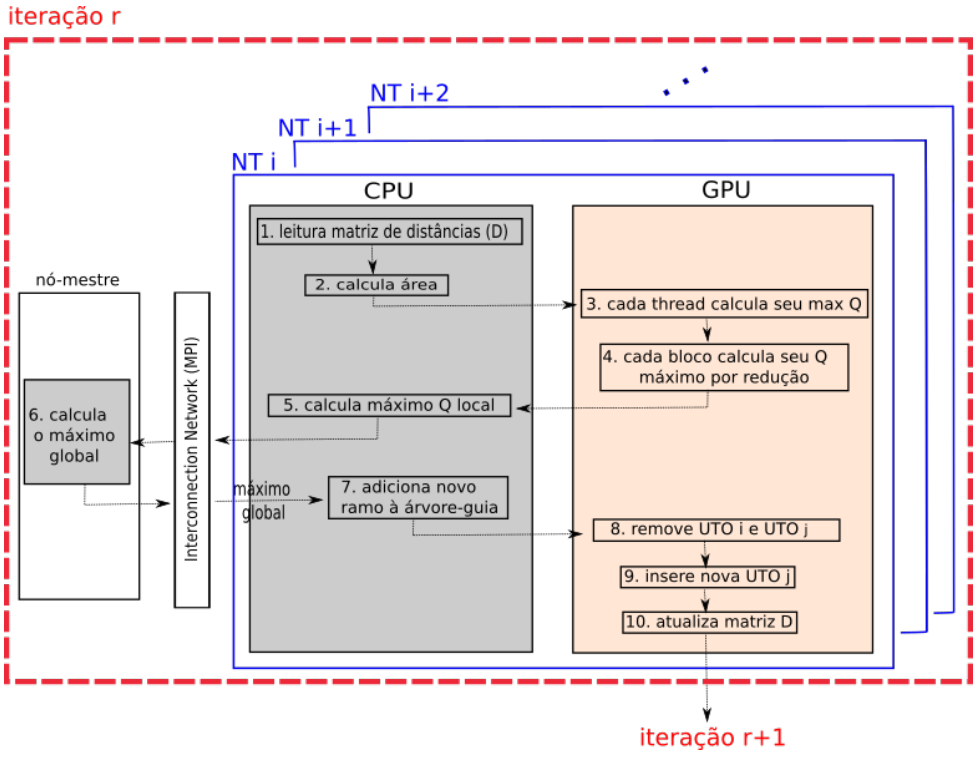

Figura 6.3: Fluxo de execução do *NJ* para a construção de uma árvore-guia em uma arquitetura híbrida *MPI-CPU-GPU*. *NT* denota um nó de trabalho.

Para iniciar a execução paralela do *NJ*, a matriz D é copiada por cada nó de trabalho para a memória global de sua *GPU*. A distribuição de tarefas desta etapa da heurística é coordenada de forma autônoma e sincronizada a cada iteração pelos nós de trabalho. Desta forma, o nó-mestre não exerce qualquer influência sobre os dados ou distribuição de trabalho, porém é informado sobre o resultado de cada iteração para que possa também conhecer a árvore-guia construída ao final da execução desta etapa, como ilustrado na Figura 6.3.

Devido às caracteristicas de agrupamento apresentado pelo *NJ*, o número de *UTOs* a cada iteração do algoritmo é reduzido. Esta redução exige que o nível de paralelismo seja reduzido de acordo, para evitar desperdício de recursos e possíveis esperas por sincronismo de nós ociosos ou com pouco trabalho a ser realizado. Seja k^l o número de *UTOs* presentes na matriz D^l, e seja p^l o número de nós de trabalho participantes durante a iteração l do *NJ*. O número de nós de trabalho participantes durante a iteração $l+1$ pode ser reduzido devido à diminuição de *UTOs* ativas em D^{l+1}, caso $\frac{k^{l+1}}{p^{l+1}}$ seja menor que um determinado limiar parametrizável para a quantidade mínima de processamento por cada nó de trabalho.

Para paralelizar o *NJ*, as iterações do algoritmo são subdivididas em três subetapas: a primeira consiste do cálculo da soma das pontuações de cada *UTO* ativa da matriz D^l; a segunda é a seleção de um par de *UTOs* que deve ser agrupado na iteração; e por fim, o cálculo da matriz D^{l+1} a partir da remoção do par de *UTOs* selecionadas, e da inserção de uma nova *UTO* representando o agrupamento deste par.

A cada iteração, o cálculo da matriz Q^l, descrita pela Equação 4.2, é dividido entre os nós de trabalho de forma autônoma, tal que cada nó seja responsável pelo cálculo de uma área semelhante da matriz. Como a matriz Q necessita ser recalculada em sua completude a cada iteração, cada nó é responsável por encontrar o melhor candidato para agrupamento entre os pares de *UTOs* em sua área de atuação sobre Q^l. Para calcular a matriz Q em uma *GPU*, cada nó utiliza um número de *threads* proporcional à area total da matriz Q pelo qual o nó é responsável. Cada *thread* pode calcular Q de mais de um único par, e deve manter o melhor dentre eles. Em seguida, realiza-se uma operação de *reduce* entre os pares com melhor Q de cada *thread*, para que o melhor par candidato local de um nó seja encontrado. Por fim, realiza-se uma operação de *all-reduce* entre os pares candidatos de cada nó de trabalho, encontrando-se, assim, o melhor par global que deve ser agrupado na iteração l.

Por fim, todos os nós calculam D^{l+1} localmente, realizando a junção entre as *UTOs* selecionadas globalmente. Este cálculo é realizado com a simples desativação de uma linha e coluna de D^l e utilizando-se de espaço liberado

por uma das *UTOs* a serem agrupadas para o armazenamento dos valores de similaridades da nova *UTO* criada.

Devido à presença do limiar para o número de nós participantes em cada passo do *NJ*, a execução do algoritmo será sempre limitada a um único nó e, consequentemente, a uma única *GPU* durante as últimas iterações do algoritmo. De forma semelhante, apenas uma *GPU* será utilizada caso k inicial seja muito pequeno.

A árvore-guia é representada como uma lista contígua de objetos que indicam o pai e os filhos de cada nó da árvore. A cada iteração do *NJ* um nó é adicionado à árvore representando a nova *UTO*, com ponteiros para as *UTOs* agrupadas representadas com filhos. A árvore-guia jamais é enviada à *GPU* e, ao final da execução desta etapa, deve estar construída e ser conhecida pelo nó-mestre.

6.4 Etapa 3: Alinhamento Progressivo

A terceira e última etapa da heurística, é a responsável por construir um alinhamento global com todas as sequências fornecidas como entrada. Este alinhamento é realizado utilizando a árvore-guia, construída na etapa anterior, para determinar a ordem em que as sequências e *UTOs* devem ser alinhadas, como ilustrado na Figura 6.4. Para a solução desta etapa, utilizamos o algoritmo de Myers-Miller, conforme descrito na Seção 4.3.

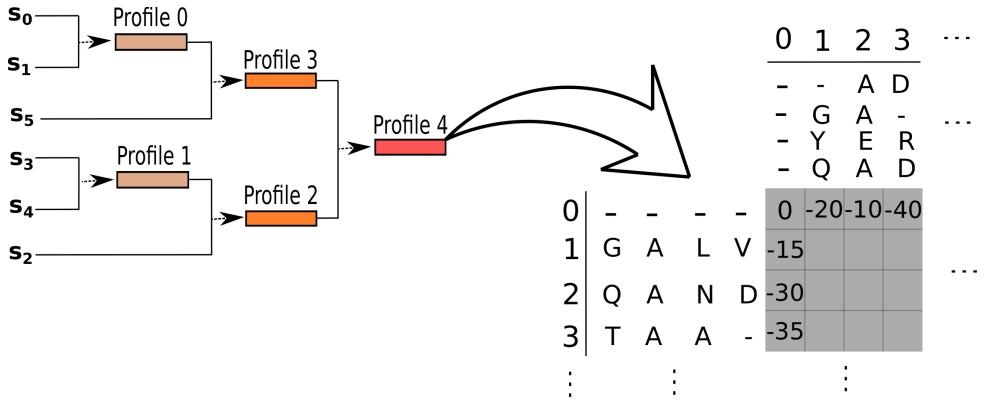

Figura 6.4: Algoritmo para o alinhamento progressivo entre 6 sequências. Nesta figura, o último alinhamento envolve duas *UTOs* compostas por 3 sequências cada uma.

Para esta etapa, o nó-mestre é responsável pela construção de um plano de execução que coordenará os nós de trabalho para a produção do alinhamento final, o resultado esperado do método heurístico. A concepção deste plano visa minimizar a quantidade de dados transferidos após cada alinhamento e maximizar a utilização de recursos de cada nó de trabalho, e utiliza a

quantidade e o comprimento das sequências das *UTOs* como uma métrica de recursos necessários para um alinhamento.

Um alinhamento pode ser visto como simples inserções de *gaps* entre os símbolos das *UTOs* participantes. Isto é, os símbolos de cada sequência em uma *UTO* sempre aparecerão em suas ordens originais após um alinhamento, com a possibilidade da existência de zero ou mais *gaps* entre dois símbolos originalmente consecutivos. Devido a esta característica, podemos representar um alinhamento simplesmente pela quantidade de *gaps* inseridos entre cada par de símbolos de cada sequência envolvida no alinhamento.

Esta representação permite que manipulações e alocações de memória adicionais não sejam necessárias durante a construção de um alinhamento, e que os alinhamentos sendo gerados, por sua vez, sejam compartilhados entre diferentes nós de trabalho de maneira eficiente, sem a necessidade de compartilhamento repetitivo dos símbolos de cada sequência. Contudo, esta simplificação traz consigo maior complexidade para iterar sobre as sequências de um alinhamento.

Conforme o plano de execução traçado pelo nó-mestre, cada alinhamento é realizado por um único nó de trabalho. Não obstante, alinhamentos entre *UTOs* já concluídas, que não dependem do resultado de alinhamentos ainda pendentes, podem ser executados paralelamente em nós diferentes.

Cada alinhamento é construído utilizando-se o algoritmo de Myers-Miller, descrito na Seção 4.3. O paralelismo em *GPUs* deste algoritmo é implementado utilizando-se a mesma estratégia descrita na Seção 6.2 para o algoritmo *LazyRScan-mNW*, onde cada iteração proposta por Myers e Miller é calculada utilizando-se frentes de execução para encontrar o ponto central de um alinhamento. Após a primeira iteração, cada execução recursiva do algoritmo pode ser paralelizada utilizando-se maiores quantidades de blocos de *threads* na *GPU*. Como o *LazyRScan-mNW* consiste de uma implementação paralela para o algoritmo de Needleman-Wunsch, nesta etapa alinhamentos entre *UTOs* compostas por mais de uma única sequência requerem adaptações à recorrência para o preenchimento da matriz D, originalmente descrito pela Equação 4.1. Seja uma *UTO* a um conjunto de sequências previamente alinhadas. A Equação 6.2 descreve a recorrência para o preenchimento da matriz D no contexto de alinhamento de duas *UTOs*. A inserção de um *gap* ocorre em todas as sequências de uma *UTO*.

$$D_{i,j} = max \begin{cases} D_{i-1,j-1} + \sum_{s_1 \in a_1} \sum_{s_2 \in a_2} \gamma_{s_1[i] \to s_2[j]} & \text{Alinhar símbolos em } \textit{UTOs} \\ D_{i-1,j} + |a_1|\, g & \text{Símbolo } \textit{gap} \text{ na } \textit{UTO } a_1 \\ D_{i,j-1} + |a_2|\, g & \text{Símbolo } \textit{gap} \text{ na } \textit{UTO } a_2 \end{cases} \tag{6.2}$$

Após a conclusão da execução por todos os nós de trabalho, o alinhamento resultante é enviado ao nó-mestre, para que seja exportado, formatado e salvo de acordo com as configurações informadas pelo usuário.

CAPÍTULO

7

Experimentos

Neste capítulo apresentamos uma comparação de resultados obtidos com nossa implementação, chamada *Museqa* (acrônimo para *Multiple Sequence Aligner*), com o *ClustalW-MPI* [Li, 2003] e o *ClustalΩ* [Sievers et al., 2011]. Estes alinhadores são a nossa base de comparação devido à sua popularidade e grande quantidade de citações de ambos na literatura. Nos experimentos utilizamos bases de sequências biológicas obtidas do *National Center for Biotechnology Information* (*NCBI*) [Coordinators, 2015], contendo sequências relacionadas aos vírus da Zika, da Dengue e do SARS-CoV-2. Também foram consideradas bases de dados sintéticas geradas aleatoriamente, identificadas por Syn20k, Syn30k e Syn40k. Estas bases sintéticas foram utilizadas para testar o comportamento da solução proposta com um grande número de sequências. Utilizamos medidas de tempo para comparação entre os métodos. A Tabela 7.1 apresenta o número e o tamanho das sequências de cada base.

Base	Número de sequências	Comprimento médio
NCBI Zika	700	3423
NCBI Dengue	6123	3392
NCBI SarS-CoV-2	7631	7096
Syn20k	20000	200
Syn30k	30000	200
Syn40k	40000	200

Tabela 7.1: Características das bases de sequências *NCBI* e sintéticas consideradas para os experimentos deste trabalho. Para as sequências sintéticas, não existe variação no comprimento das sequências, de forma que todas possuem exatamente o mesmo número de caracteres.

Neste capítulo, todos os tempos exibidos como resultado nas tabelas adi-

ante são referentes a uma média de 3 execuções de cada método.

7.1 Ambiente de Testes

Ainda que o *MPI* suporte a execução em *clusters* conectados por uma rede *Ethernet*, os casos de teste foram executados em um ambiente composto por um computador com processador Intel® Xeon® CPU E5-1620 v2 @ 3.50GHz de 8 núcleos, 64GB de RAM, sistema operacional Ubuntu, equipado de 4 GPUs Nvidia® GeForce® GTX 1080 Ti em que cada *GPU* possui 11GB memória e um total de 3584 *CUDA cores*, distribuídos em 20 *SMs*.

7.2 Desempenho dos Métodos

Para analisar o desempenho dos métodos propostos, comparamos a nossa proposta à última versão estável disponível do *ClustalW-MPI* e do *ClustalΩ*. A Tabela 7.2 exibe os tempos de execução do primeiro estágio do método heurístico, o alinhamento par-a-par, para diferentes configurações de execução de cada implementação. Para o *ClustalW-MPI* utilizados execuções com 4 e 8 nós de trabalho como base de comparação, e para o *ClustalΩ* execução com 8 nós de trabalho. Para a nossa proposta, devido ao número de *GPUs* disponíveis no ambiente de testes, a quantidade máxima de nós de trabalho utilizados em cada execução é limitada a 4.

	ClustalW-MPI		ClustalΩ	Museqa			Speedups	
Base	4	(a) 8	(b) 8	1	2	(c) 4	a/c	b/c
Zika	3h52m	2h10m	31s	50s	26s	14s	557	2,2
Dengue	212h27m	196h7m	8m43s	1h2m	33m	18m	654	0,5
SarS-CoV-2	690h20m	597h16m	1h27m	7h29m	3h25m	1h53m	317	0,8
Syn20k	5h04m	2h51m	32s	7m7s	3m38s	1m55s	63	0,3
Syn30k	8h15m	6h24m	51s	16m16s	8m23s	4m36s	83	0,2
Syn40k	13h15m	11h28m	1m21s	31m34s	16m34s	8m54s	77	0,2

Tabela 7.2: Comparação entre os resultados obtidos pelo *ClustalW-MPI*, *ClustalΩ* e a nossa proposta, em relação à primeira etapa do método heurístico, o alinhamento par-a-par. Nesta tabela, os métodos são comparados com todas as bases de sequências e com diferentes números de nós de trabalho.

Os resultados obtidos mostram que nossa proposta apresenta visível escalabilidade para esta etapa, em relação à quantidade de nós de trabalho utilizados. Nota-se, também, que os *speedups* maiores são obtidos com bases compostas por sequências de maiores comprimentos, demonstrando a eficiência do paralelismo de granulidade fina disponível com a utilização de *GPUs*.

É necessário relembrar que o *ClustalΩ* utiliza algoritmos distintos para esta etapa em relação àqueles implementados pelo *ClustalW-MPI* e a nossa proposta, de forma que a distribuição de trabalho entre as diferentes etapas da heurística é também modificada pela implementação do *ClustalΩ*.

A Figura 7.1 demonstra escalabilidade quase linear entre o número de nós de trabalho e o tempo de execução de nossa proposta.

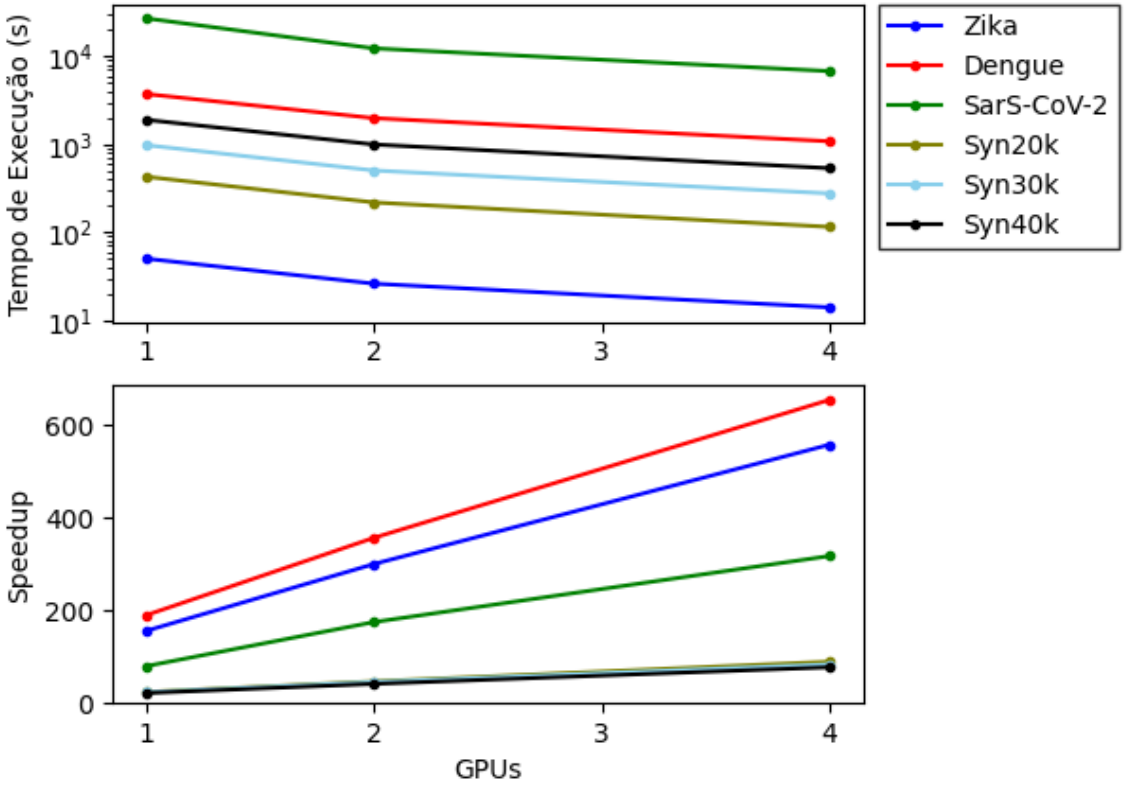

Figura 7.1: Tempo de execução e speedup obtido de cada base de sequências testada em relação à quantidade de nós de trabalho utilizados, conforme mostrado na Tabela 7.2, para a primeira etapa de alinhamento par-a-par. Nesta imagem, utiliza-se a escala logarítmica para o eixo de tempo. O speedup é calculado em comparação à execução com 8 nós de trabalho do *ClustalW-MPI*.

Por necessitar de paralelismo de granulosidade fina, a implementação atual do *ClustalW-MPI* não apresenta uma versão paralela para a segunda etapa do método heurístico utilizado, para a construção da árvore-guia. Por esta razão, o algoritmo *NJ* implementado pelo *ClustalW-MPI* não apresenta desempenho superior à sua execução sequencial [Li, 2003] mesmo que mais de um nó de trabalho seja utilizado. A Tabela 7.3 mostra a comparação entre os tempos médios de execução desta etapa para cada base de sequências no ambiente de testes supracitado.

Para a segunda etapa, nota-se novamente a escalabilidade de nossa solução em relação à quantidade de nós de trabalho disponíveis para execução.

Base	ClustalW-MPI		ClustalΩ	Museqa			Speedups	
	4	(*a*) 8	(*b*) 8	1	2	(*c*) 4	*a*/*c*	*b*/*c*
Zika	14s	15s	14s	0.18s	0.12s	0.11s	136	127
Dengue	36m27s	37m18s	2m21s	12s	11s	11s	203	13
SarS-CoV-2	1h7m	1h8m	40m41s	22s	19s	18s	226	136
Syn20k	35h10m	35h10m	7s	4m53s	4m4s	3m44s	565	0,01
Syn30k	165h35m	165h35m	11s	15m31s	12m44s	11m35s	858	0,01
Syn40k	385h	385h	15s	35m40s	29m33s	25m33s	834	0,01

Tabela 7.3: Comparação entre os resultados obtidos pelo *ClustalW-MPI*, *ClustalΩ* e a nossa proposta, em relação à segunda etapa do método heurístico, referente à construção da árvore-guia. Nesta tabela, os métodos são comparados com todas as bases de sequências e com diferentes números de nós de trabalho.

Contudo, a escalabilidade do método torna-se mais clara durante a execução de bases com uma quantidade maior de sequências. Isto é esperado devido à diminuição de paralelismo, realizado por nossa implementação, quando o número de *UTOs* na árvore-estrela do algoritmo *NJ* cai abaixo do limiar configurado para a execução. A Figura 7.2 demonstra esta escabilidade obtida com o aumento de disponibilidade de recursos para a execução de nossa solução.

É importante ressaltar, novamente, que o algoritmo utilizado pelo *ClustalΩ* em sua segunda etapa é diferente daqueles utilizados pelo *ClustalW-MPI* e a nossa implementação. Esta diferença pode ser percebida na Tabela 7.3 à semelhança do que ocorre na primeira etapa como exposto na Tabela 7.2.

Conforme os dados apresentados nas Tabelas 7.2 e 7.3 acima, nota-se que o tamanho das sequências influencia significativamente sobre os tempos de execução da primeira etapa. Isto fica claro ao observar-se que, de forma consistente, as bases de sequências sintéticas – Syn20k, Syn30k e Syn40k – gastam menos tempo que as bases provenientes do *NCBI* mesmo que estas últimas possuam números de sequências entre 3 e 57 vezes menores que as primeiras. Também é visível que o número de nós de trabalho empregados afetam de forma inversamente proporcional o tempo de execução desta etapa.

Por outro lado, o único fator relevante para o tempo consumido pela segunda etapa é a quantidade de sequências. Estas observações são consistentes com os dados apresentados por Liu [Liu et al., 2009], dado que as complexidades dos algoritmos utilizados nessas etapas são $O(k^2 l_{avg}^2)$ e $O(k^3)$ respectivamente, onde l_{avg} é o tamanho médio das sequências e k o número de sequências.

Em relação à terceira e última etapa, para a construção progressiva do alinhamento múltiplo resultante entre as sequências, a Tabela 7.4 exibe que os resultados obtidos por nossa implementação, em comparação ao *ClustalW-*

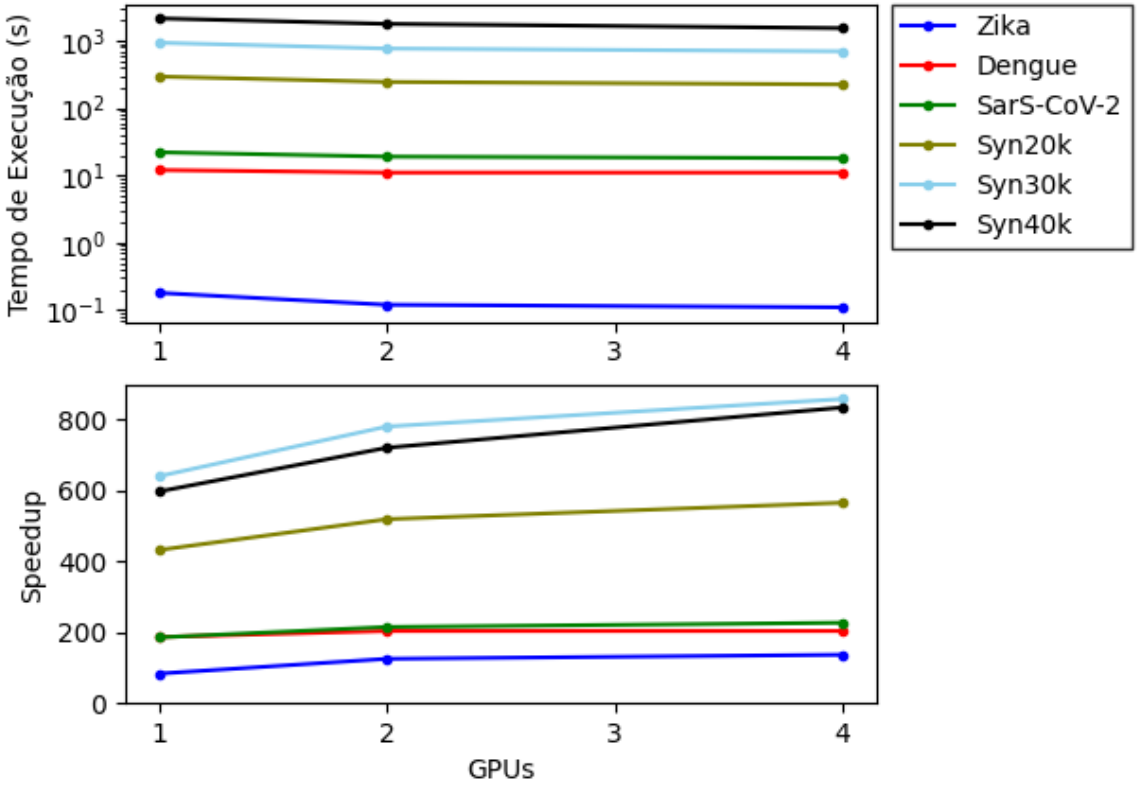

Figura 7.2: Tempo de execução para a etapa de construção da árvore-guia de cada base de sequências testada em relação à quantidade de nós de trabalho utilizados, conforme mostrado na Tabela 7.3. Nesta imagem, a base de comparação para os speedups é a execução com 8 nós de trabalho do *ClustalW-MPI*.

Base	ClustalW-MPI		ClustalΩ	Museqa			Speedups	
	4	(a) 8	(b) 8	1	2	(c) 4	a/c	b/c
Zika	3m32s	2m53s	16m34s	3m47s	2m5s	1m15s	2,3	13,3
Dengue	34m45s	28m59s	2h27m	41m56s	24m10s	12m54s	2,2	11,4
SarS-CoV-2	35m57s	28m58s	14h24m	55m13s	29m22s	15m34s	1,9	55,5
Syn20k	14h57m	8h55m	8h5m	7h39m	4h10m	2h17m	3,9	2,7
Syn30k	50h25m	30h1m	10h45m	26h18m	15h3m	8h11m	3,7	1,3
Syn40k	151h	90h	16h	52h1m	28h6m	15h29m	5,8	1,1

Tabela 7.4: Comparação entre os resultados obtidos pelo *ClustalW-MPI*, *ClustalΩ* e a nossa proposta, em relação à terceira etapa do método heurístico, referente ao alinhamento progressivo entre sequências. Nesta tabela, os métodos são comparados com todas as bases de sequências e com diferentes números de nós de trabalho.

MPI apresenta *speedups* de até 5,8 e de até 55,5 se comparado ao *ClustalΩ*. Novamente, os resultados apresentados consistem de uma média de tempo entre 3 execuções em cada configuração.

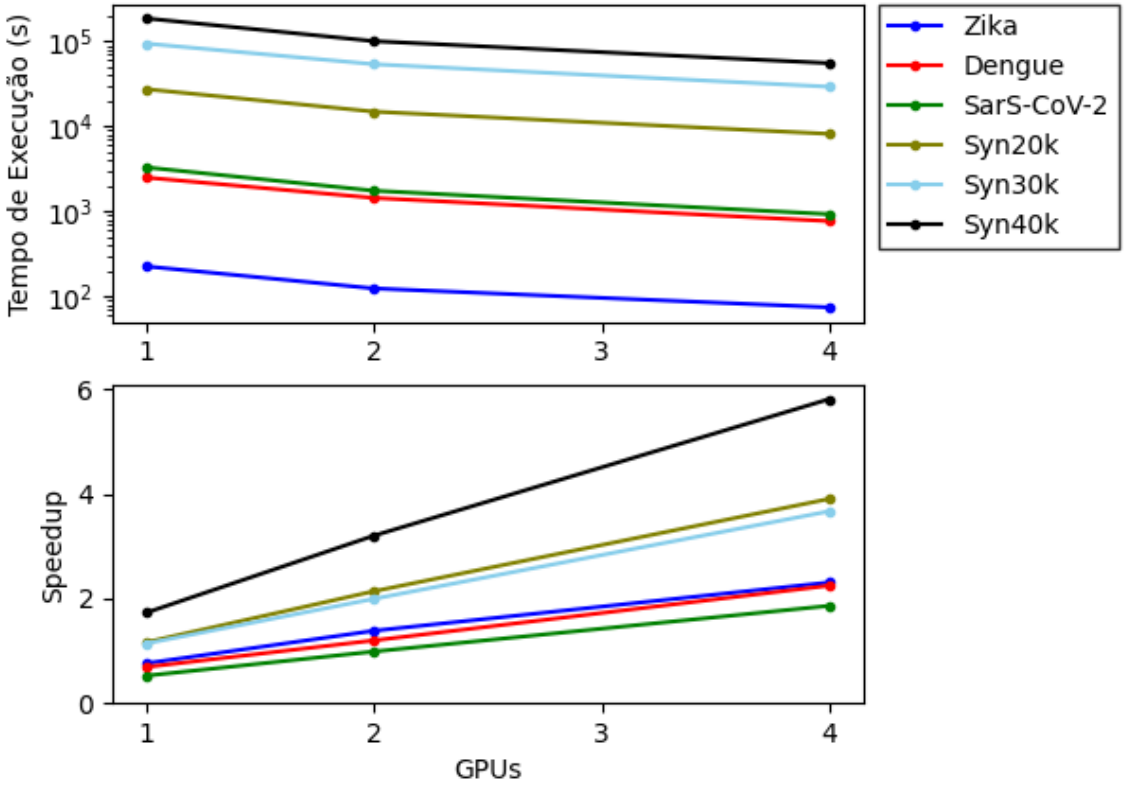

Figura 7.3: Tempo de execução da etapa final para a construção do alinhamento progressivo entre as sequências de cada base testada em relação à quantidade de nós de trabalho utilizados, conforme mostrado na Tabela 7.4. Nesta imagem, utiliza-se a escala logarítmica para o eixo de tempo de execução.

Como visto nos resultados apresentados na Tabela 7.4, a versão atual de nossa implementação ainda não apresenta grandes *speedups* e escalabilidade para a última etapa do método heurístico. Ainda assim, a Figura 7.3 mostra que, apesar de o *speedup* não escalar rapidamente, nossa implementação apresenta melhorias de tempo em todos os casos testados. Nota-se também que a nossa implementação e o *ClustalW-MPI* demonstram grande sensibilidade à quantidade total de sequências a serem alinhadas nesta etapa.

A Tabela 7.5 mostra a soma com o tempo total de execução de nossa solução, em comparação ao *ClustalW-MPI* utilizando 8 nós de trabalho, apresenta *speedups* entre 19 e 380, considerando-se todas as três etapas do método heurístico apresentado neste trabalho. Já em relação ao *ClustalΩ* os *speedups* variam até 11,7.

É relevante notar que a etapa para alinhamento par-a-par domina o tempo consumido por todas as bases de sequências obtidas do *NCBI*, enquanto que, para as bases sintéticas, o tempo das outras duas etapas torna-se relevantes e dominantes em relação ao tempo total de execução. Este fato é determinante

Base	ClustalW-MPI		ClustalΩ	Museqa			Speedups	
	4	(*a*) 8	(*b*) 8	1	2	(*c*) 4	*a*/*c*	*b*/*c*
Zika	3h52m	2h13m	17m19s	4m37s	2m31s	1m29s	89,6	11,7
Dengue	213h38m	197h13m	2h38m	1h44m	57m21s	31m5s	380	5,1
SarS-CoV-2	691h44m	598h52m	16h31m	8h24m	4h21m	2h9m	278	7,7
Syn20k	55h11m	46h56m	8h5m	7h51m	4h17m	2h22m	19,8	3,4
Syn30k	224h15m	202h	10h46m	26h50m	15h23m	8h27m	23,9	1,3
Syn40k	549h15m	486h28m	16h1m	53h8m	28h52m	16h3m	30,3	1,0

Tabela 7.5: Comparação entre a soma dos resultados obtidos em todas as etapas pelo *ClustalW-MPI*, *ClustalΩ* e a nossa proposta.

para a comparação entre os métodos em relação ao tempo total de execução para os casos com números maiores de sequências.

CAPÍTULO

8

Conclusão

O alinhamento de várias sequências biológicas é uma das mais importantes ferramentas 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 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 heurísticas. Aliados a computação de alto desempenho, existem diversas propostas para acelerar a execução desta tarefa. Algumas propostas paralelizam apenas a primeira etapa do alinhamento progressivo, enquanto outras paralelizam apenas a segunda ou terceira etapa. Além de implementações parciais, na literatura encontram-se normalmente implementações com apenas um tipo de paralelismo: somente granularidade grossa, com ferramentas como *MPI* [Gropp et al., 1999]; ou somente granularidade fina, em arquiteturas como *CUDA* [Cook, 2012].

Neste trabalho descrevemos o problema do alinhamento de várias sequências, implementações já existentes e as possíveis soluções para este problema. Descrevemos, também, diferentes arquiteturas para aplicações paralelas, como *clusters* de computadores e *GPUs*. Apresentamos a heurística de Alinhamento Progressivo e todas as suas etapas, descrevendo alternativas de algoritmos sequenciais e paralelos para cada uma delas. Descrevemos os algoritmos de Needleman-Wunsh, Neighbor-Joining, e Myers-Miller, além de propormos formas de implementação paralela para cada um desses algoritmos.

Implementamos versões em paralelismo híbrido para cada uma das etapas da heurística de Alinhamento Progressivo, obtendo resultados expressivos em cada uma das etapas individualmente, e *speedups* de até 256 vezes

no somatório de todo o tempo de execução em comparação com o *ClustalW-MPI* [Li, 2003].

Dentre as oportunidades de expansão deste trabalho, encontram-se melhorias para utilização mais eficiente de memória em cada um dos algoritmos implementados, e estudos sobre a qualidade dos alinhamentos gerados e os desempenhos de cada etapa com diferentes seleções de parâmetros.

Bibliografia

[Alawneh et al., 2020] Alawneh, L., Shehab, M. A., Al-Ayyoub, M., Jararweh, Y., e Al-Sharif, A. Z. (2020). A scalable multiple pairwise protein sequence alignment acceleration using hybrid CPU–GPU approach. *Cluster Computing*. Citado na página 3.

[Alsaadi et al., 2019] Alsaadi, E. A., Neuman, B. W., e Jones, I. M. (2019). A fusion peptide in the spike protein of mers coronavirus. *Viruses*, 11:825. Citado nas páginas ix e 17.

[Altschul et al., 1990] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., e Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410. Citado nas páginas 2 e 7.

[Araujo et al., 2017] Araujo, E., Stefanes, M. A., O. Ferlete, V., e Rozante, L. C. S. (2017). Multiple sequence alignment using hybrid parallel computing. In *17th IEEE International Conference on Bioinformatics and Bioengineering*, páginas 175–180. Citado na página 3.

[Carrillo and Lipman, 1988] Carrillo, H. e Lipman, D. (1988). The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, 48(5):1073–1082. Citado nas páginas 2 e 17.

[Chan et al., 1992] Chan, S. C., Wong, A. K. C., e Chiu, D. K. Y. (1992). A survey of multiple sequence comparison methods. *Bulletin of mathematical biology*, 54(4):563–598. Citado na página 17.

[Chang et al., 2012] Chang, J.-M., Di Tommaso, P., Taly, J.-F., e Notredame, C. (2012). Accurate multiple sequence alignment of transmembrane proteins with psi-coffee. *BMC Bioinformatics*, 13(4). Citado na página 9.

[Chen et al., 2017] Chen, X., Wang, C., Tang, S., Yu, C., e Zou, Q. (2017). CMSA: a heterogeneous CPU/GPU computing system for multiple similar

RNA/DNA sequence alignment. *BMC Bioinformatics*, 18(315). Citado na página 3.

[Cook, 2012] Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Elsevier. Citado nas páginas ix, 30, 31, e 53.

[Coordinators, 2015] Coordinators, N. R. (2015). Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, 44(D1):D7–D19. Citado na página 45.

[Dayhoff et al., 1978] Dayhoff, M. O., Schwartz, R. M., e Orcutt, B. C. (1978). *Atlas of protein sequece and structure*, chapter A model of evolutionary change in proteins, páginas 345–352. National Biomedical Research Foundation, Washington, DC, 3 edition. Citado na página 16.

[de Brito, 2003] de Brito, R. T. (2003). Alinhamento de sequências biológicas. Master's thesis, Universidade de São Paulo. Citado na página 26.

[Deng et al., 2006] Deng, X., E., L., J., S., e W., C. (2006). Parallel implementation and performance characterization of muscle. In *IPDPS'06*, páginas 1–7. Citado na página 9.

[Edgar, 2004] Edgar, R. C. (2004). Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797. Citado nas páginas 2, 9, e 20.

[Felsenstein, 1981] Felsenstein, J. (1981). Evolutionary trees from dna sequences: a maximum likelihood approach. *J Mol Evol*, 17(6):368–376. Citado na página 22.

[Feng and Doolittle, 1987] Feng, D.-F. e Doolittle, R. F. (1987). Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J. Mol. Evol.*, 25(4):351–360. Citado nas páginas 2, 17, 19, e 27.

[Ferlete and Stefanes, 2015] Ferlete, V. O. e Stefanes, M. A. (2015). Alinhamento de várias sequências utilizando arquiteturas paralelas híbridas. Master's thesis, Universidade Federal de Mato Grosso do Sul, Campo Grande, MS, Brasil. Citado na página 10.

[Gropp et al., 1999] Gropp, W., Lusk, E., e Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press. Citado nas páginas 29, 31, e 53.

[Guindon and Gascuel, 2003] Guindon, S. e Gascuel, O. (2003). A simple, fast, and accurate algorithm to estimate large phylogenies by maximum likelihood. *Systematic Biology*, 52(5):696–704. Citado na página 22.

[Henikoff, 1992] Henikoff, J. (1992). Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci*, 89(22):10915–10919. Citado na página 16.

[Higgins and Sharp, 1988] Higgins, D. G. e Sharp, P. M. (1988). Clustal: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244. Citado nas páginas 8 e 22.

[Johnson et al., 2010] Johnson, L. S., Eddy, S. R., e Portugaly, E. (2010). Hidden markov model speed heuristic and iterative hmm search procedure. *BMC Bioinformatics*, 11(1):431. Citado na página 3.

[Katoh et al., 2002] Katoh, K., Misawa, K., Kuma, K., e Miyata, T. (2002). MAFFT: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic Acids Research*, 30(14):3059–3066. Citado nas páginas 2 e 9.

[Katoh and Toh, 2007] Katoh, K. e Toh, H. (2007). Parttree: an algorithm to build an approximate tree from a large number of unaligned sequences. *Bioinformatics*, 23(3):372–374. Citado na página 9.

[Katoh and Toh, 2010a] Katoh, K. e Toh, H. (2010a). Parallelization of the MAFFT multiple sequence alignment program. *Bioinformatics*, 26(15):1899–1900. Citado na página 3.

[Katoh and Toh, 2010b] Katoh, K. e Toh, H. (2010b). Parallelization of the mafft multiple sequence alignment program. *Bioinformatics*, 26(15):1899–1900. Citado na página 10.

[Khare et al., 2007] Khare, N., Khare, A., e Khan, F. (2007). HCudaBLAST: an implementation of BLAST on hadoop and CUDA. *Journal of Big Data*, 4:41. Citado na página 7.

[Lan et al., 2016] Lan, H., Chan, Y., Xu, K., Schmidt, B., Peng, S., e Liu, W. (2016). Parallel algorithms for large-scale biological sequence alignment on Xeon-Phi based clusters. *BMC bioinformatics*, 17(9):267. Citado na página 3.

[Lassmann, 2019] Lassmann, T. (2019). Kalign 3: multiple sequence alignment of large datasets. *Bioinformatics*, 36(6):1928–1929. Citado na página 9.

[Lassmann and Sonnhammer, 2005] Lassmann, T. e Sonnhammer, E. (2005). Kalign: An accurate and fast multiple sequence alignment algorithm. *BMC bioinformatics*, 6:298. Citado nas páginas 2 e 8.

[Levenshtein, 1966] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10:707–710. Citado na página 14.

[Li, 2003] Li, K.-B. (2003). Clustalw-mpi: Clustalw analysis using distributed and parallel computing. *Bioinformatics*, 19(12):1585–1586. Citado nas páginas 3, 4, 8, 45, 47, e 54.

[Liu et al., 2009] Liu, Y., Schmidt, B., e Maskell, D. L. (2009). MSA-CUDA: Multiple sequence alignment on graphics processing units with CUDA. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, páginas 121–128. Citado nas páginas 8, 22, 39, e 48.

[Lopes et al., 2012] Lopes, H. S., Lima, C. R. E., e Moritz, G. L. (2012). A parallel algorithm for large-scale multiple sequence alignment. *Computing and Informatics*, 29(6+):1233–1250. Citado na página 3.

[M. Cooper and Brown, 2008] M. Cooper, G. e Brown, C. (2008). Qualifying the relationship between sequence conservation and molecular function. *Genome Res.*, 18:201–205. Citado na página 12.

[Masuno et al., 2007] Masuno, S., Maruyama, T., Yamaguchi, Y., e Konagaya, A. (2007). An fpga implementation of multiple sequence alignment based on carrillo-lipman method. *2007 International Conference on Field Programmable Logic and Applications*, páginas 489–492. Citado nas páginas ix e 18.

[Mount, 2004] Mount, D. W. (2004). *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2nd edition. Citado na página 1.

[Myers and Miller, 1988] Myers, E. W. e Miller, W. (1988). Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17. Citado nas páginas ix, 22, 27, e 28.

[Needleman and Wunsch, 1970] Needleman, S. B. e Wunsch, C. D. (1970). 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. Citado nas páginas 1, 16, 21, 22, 23, e 37.

[Ng and Henikoff, 2001] Ng, P. C. e Henikoff, S. (2001). Predicting deleterious amino acid substitutions. *Genome Res.*, 11(5):863–874. Citado nas páginas 1 e 12.

[Notredame and Higgins, 1996] Notredame, C. e Higgins, D. G. (1996). Saga: Sequence alignment by genetic algorithm. *Nucleic Acids Research*, 24(8):1515–1524. Citado na página 3.

[Notredame et al., 2000] Notredame, C., Higgins, D. G., e Heringa, J. (2000). T-coffee: a novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205 – 217. Citado nas páginas 9 e 20.

[Ogden and Rosenberg, 2006] Ogden, T. H. e Rosenberg, M. S. (2006). Multiple Sequence Alignment Accuracy and Phylogenetic Inference. *Systematic Biology*, 55(2):314–328. Citado na página 22.

[Price et al., 2010] Price, M. N., Dehal, P. S., e Arkin, A. P. (2010). Fasttree 2 – approximately maximum-likelihood trees for large alignments. *PLOS ONE*, 5(3):1–10. Citado na página 22.

[Saitou and Nei, 1987a] Saitou, N. e Nei, M. (1987a). The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 4(4):406–425. Citado na página 3.

[Saitou and Nei, 1987b] Saitou, N. e Nei, M. (1987b). The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425. Citado nas páginas 21, 25, e 39.

[Sievers et al., 2011] Sievers, F., Wilm, A., Dineen, D., TJa, G., Karplus, K., Li, W., Lopez, R., McWilliam, H., Remmert, M., Söding, J., Thompson, J., e Higgins, D. (2011). Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega. *Molecular Systems Biology*, 539(7). Citado nas páginas 4, 8, e 45.

[Simonsen et al., 2008] Simonsen, M., Mailund, T., e Pedersen, C. (2008). Rapid neighbour-joining. volume 5251, páginas 113–122. Citado na página 21.

[Smith and Waterman, 1981] Smith, T. F. e Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197. Citado na página 1.

[Sokal and Michener, 1958] Sokal, R. R. e Michener, C. D. (1958). A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38:1409–1438. Citado na página 21.

[Sonnhammer et al., 1998] Sonnhammer, E. L. L., Eddy, S. R., Birney, E., Bateman, A., e Durbin, R. (1998). Pfam: Multiple sequence alignments and hmm-profiles of protein domains. *Nucleic Acids Research*, 26(1):320–322. Citado na página 10.

[Thompson et al., 1997] Thompson, J. D., Gibson, T. J., Plewniak, F., Jeanmougin, F., e Higgins, D. G. (1997). The clustal-x windows interface: Flexible strategies for multiple sequence alignment aided by quality analysis tools. *Nucleic Acids Research*, 25(24):4876–4882. Citado na página 8.

[Thompson et al., 1994] Thompson, J. D., Higgins, D. G., e Gibson, T. J. (1994). 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. Citado nas páginas 2, 8, e 20.

[Truong et al., 2014] Truong, H., Li, D., Sajjapongse, K., Conant, G., e Becchi, M. (2014). Large-scale pairwise alignments on GPU clusters: Exploring the implementation space. *J. Sign. Process. Syst.*, 77:131–149. Citado nas páginas 3, 10, e 37.

[Wang and Jiang, 1994] Wang, L. e Jiang, T. (1994). On the complexity of multiple sequence alignment. *J. Comp. Biol*, páginas 337–348. Citado nas páginas 2 e 17.

[Wheeler, 2009] Wheeler, T. J. (2009). Large-scale neighbor-joining with ninja. In Salzberg, S. L. e Warnow, T., editors, *Algorithms in Bioinformatics*, páginas 375–389, Berlin, Heidelberg. Springer Berlin Heidelberg. Citado na página 21.

[Wilm et al., 2008] Wilm, A., Higgins, D. G., e Notredame, C. (2008). R-coffee: a method for multiple alignment of non-coding rna. *Nucleic Acids Research*, 36(9):e52–e52. Citado na página 9.

[Yoon, 2009] Yoon, B.-J. (2009). Hidden markov models and their applications in biological sequence analysis. *Current Genomics*, 10. Citado na página 19.

[Zambrano-Vega et al., 2017] Zambrano-Vega, C., Nebro, A. J., García-Nieto, J., e Aldana Montes, J. F. (2017). M2Align: parallel multiple sequence alignment with a multi-objective metaheuristic. *Bioinformatics*. Citado na página 3.

[Zhang et al., 2019] Zhang, C., Zheng, W., Mortuza, S. M., Li, Y., e Zhang, Y. (2019). Deepmsa: constructing deep multiple sequence alignment to improve contact prediction and fold-recognition for distant-homology proteins. *Bioinformatics*, 36(7):2105–2112. Citado nas páginas 2 e 10.

[Zhao and Chu, 2014] Zhao, K. e Chu, X. (2014). G-blastn: accelerating nucleotide alignment by graphics processors. *Bioinformatics*, 30(10):1384–1391. Citado na página 7.

[Zheng et al., 2012] Zheng, R., Zhang, Q., Jin, H., Shao, Z., e Feng, X. (2012). Parallelization mechanisms of neighbor-joining for CUDA enabled devices. *2012 ChinaGrid Annual Conference*, 32(7):182–188. Citado na página 39.

[Zola et al., 2007] Zola, J., Yang, X., Rospondek, A., e Aluru, S. (2007). Parallel T-Coffee: A parallel multiple sequence aligner. In *Proceedings of ISCA PDCS-2007*, páginas 248–253. Citado na página 9.

[Zomaya, 2006] Zomaya, A. Y. (2006). *Handbook of nature-inspired and innovative computing: integrating classical models with emerging technologies*. Springer Science and Business Media. Citado na página 16.

[Zou et al., 2019] Zou, H., Tang, S., Yu, C., Fu, H., Li, Y., e Tang, W. (2019). ASW: Accelerating smith–waterman algorithm on coupled CPU–GPU architecture. *International Journal of Parallel Programming*, 47:388–402. Citado na página 3.

[Zou et al., 2015] Zou, Q., Hu, Q., Guo, M., e Wang, G. (2015). HAlign: Fast multiple similar DNA/RNA sequence alignment based on the centre star strategy. *Bioinformatics*, 31(15):2475–2481. Citado na página 3.