Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Alinhamento Primário e Secundário de Sequências Biológicas em Arquiteturas de Alto Desempenho

Daniel Sundfeld Lima

Brasília
2017

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Alinhamento Primário e Secundário de Sequências Biológicas em Arquiteturas de Alto Desempenho

Daniel Sundfeld Lima

Tese apresentada como requisito parcial
para conclusão do Doutorado em Informática

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2017

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Doutorado em Informática

Coordenador: Prof. Dr. Bruno Luiggi Macchiavello Espinoza

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora) — UnB
Prof. Dr. George Luiz Medeiros Teodoro — UnB
Prof. Dr. Edson Norberto Cáceres — UFMS
Prof.ª Dr.ª Maria Emilia Machado Telles Walter — UnB
Prof. Dr. Wellington Santos Martins — UFG

CIP — Catalogação Internacional na Publicação

Sundfeld Lima, Daniel.

Alinhamento Primário e Secundário de Sequências Biológicas em Arquiteturas de Alto Desempenho / Daniel Sundfeld Lima. Brasília : UnB, 2017.
191 p. : il. ; 29,5 cm.

Tese (Doutorado) — Universidade de Brasília, Brasília, 2017.

1. Alinhamento de Sequências, 2. A-Estrela, 3. Algoritmo de Sankoff, 4. GPU

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Alinhamento Primário e Secundário de Sequências Biológicas em Arquiteturas de Alto Desempenho

Daniel Sundfeld Lima

Tese apresentada como requisito parcial
para conclusão do Doutorado em Informática

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
UnB

Prof. Dr. George Luiz Medeiros Teodoro
UnB

Prof. Dr. Edson Norberto Cáceres
UFMS

Prof.ª Dr.ª Maria Emilia Machado Telles Walter
UnB

Prof. Dr. Wellington Santos Martins
UFG

Prof. Dr. Bruno Luiggi Macchiavello Espinoza
Coordenador do Programa em Pós-Graduação em Informática

Brasília, 19 de dezembro de 2017

Dedicatória

À Juja. Obrigado por tudo.

Agradecimentos

Agradeço a Deus, acima de tudo, pelas oportunidades que me foram dadas.

Agradeço à minha esposa Elaine Cristine Matos Sundfeld. Obrigado pelo apoio incondicional, pelas conversas e, acima de tudo, a paciência que você me ofereceu. Nós sabemos o tanto que você me estimulou para que eu pudesse estar aqui.

À minha orientadora Alba, agradeço profundamente os seus ensinamentos e o tempo dedicado a esse trabalho. Há muitos anos estamos percorrendo essa jornada e sempre me lembrarei da paciência, as contínuas reuniões e a forma incrível que você conduz os trabalhos.

Aos professores Jan Gorodkin e Jakob Hull Havgaard, que me receberam muito bem na Dinamarca, onde morei durante um ano incrível e que nunca irei esquecer.

Ao professor George, agradeço suas orientações e ajuda no desenvolvimento deste trabalho, através dos debates sobre estratégias e execuções de testes que tanto contribuíram para nossa pesquisa.

Ao amigo de laboratório Jeremias Gomes, que sempre me auxiliou em manter as máquinas do laboratório ligadas, especialmente quando eu não me encontrava por perto para poder ligá-las.

A todos os meus colegas do laboratório RTH em Copenhague e aos amigos brasileiros que fizemos por lá, muito obrigado pela companhia!

Aos meus pais, avós e minha família, eu agradeço. Vocês são os exemplos que sigo nesta vida, aqueles que me espelho para ter sucesso e busco continuamente ser motivo de orgulho para vocês.

Resumo

O alinhamento múltiplo primário de sequências biológicas é um problema muito importante em Biologia Molecular, pois permite que sejam detectadas similaridades e diferenças entre um conjunto de sequências. Esse problema foi provado NP-Completo e, por essa razão, geralmente algoritmos heurísticos são usados para resolvê-lo. No entanto, a obtenção da solução ótima é bastante desejada e, por essa razão, existem alguns algoritmos exatos que solucionam esse problema para um número reduzido de sequências. As sequências de RNA, diferente do DNA, não possuem dupla-hélice e podem dobrar-se, pois seus nucleotídeos podem formar pares de bases. É conhecido na Biologia Molecular que a função dessa estrutura está ligada à sua conformação espacial, e não à composição de seus nucleotídeos. Obter a estrutura secundária (2D) de uma sequência de RNA também exige uma grande quantidade de recursos computacionais, até mesmo para um pequeno número de sequências. Desta forma, as arquiteturas de alto desempenho são muito importantes para a obtenção dos resultados em um tempo factível. A presente tese visa investigar os problemas do alinhamento múltiplo primário e do alinhamento em pares secundário, utilizando arquiteturas de alto desempenho para acelerar a obtenção de resultados. Para o alinhamento primário ótimo de múltiplas sequências, propusemos na presente Tese o PA-Star, uma estratégia *multithreaded* baseada no algoritmo A-Star que usa uma política sensível à localidade de atribuição de trabalho às threads. De modo a lidar com o alto uso de memória, nossa estratégia PA-Star usa tanto memória RAM como disco. Para o alinhamento estrutural (2D) de sequências de RNA, propusemos o Foldalign 2.5, que é uma estratégia *multithreaded* heurística baseada no algoritmo exato de Sankoff, capaz de obter o alinhamento estrutural de grandes sequências em tempo reduzido. Finalmente, propusemos o CUDA-Sankoff, que é capaz de obter o alinhamento estrutural ótimo entre duas sequências de RNA em GPU (*Graphics Processing Unit*).

Palavras-chave: Alinhamento de Sequências, A-Estrela, Algoritmo de Sankoff, GPU

Abstract

The primary multiple sequence Alignment is a very important problem in Molecular Biology since it is able to detect similarities and differences in a set of sequences. This problem has been proven NP-Hard and, for this reason, heuristic algorithms are usually used to solve it. Nevertheless, obtaining the optimal solution is highly desirable and there are indeed some exact algorithms that solve this problem for a reduced number of sequences. The RNA sequences are different than the DNA, they do not have double helix, their nucleotides can form base pairs and the sequence can fold on itself. It is known in the Molecular Biology that, the function of the RNA is related to its spatial structure. Calculating the secondary structure of RNA sequences also demand a high amount of computational resources, even for a small number of sequences. The High Performance Computing (HPC) Platforms can be used in order to produce results faster. The current thesis aims to investigate the primary multiple sequence alignment and the secondary pairwise sequence alignment, using High Performance Architectures to accelerate and obtaining results in reasonable time. For the primary multiple sequence alignment, we propose the PA-Star, a multithreaded solution based on the A-Star algorithm using a locality sensitive hash to distribute the workload among the threads. Due to the high RAM memory usage required by the algorithm, our strategy can also uses disk. For the RNA structural alignment, we proposed the Foldalign 2.5, a multithreaded solution that uses heuristics to reduce the Sankoff Algorithm complexity, and can obtain the pairwise structural alignment of large sequences in reduced time. Finally, we proposed CUDA-Sankoff, that obtains the optimal pairwise structural alignment for RNA sequences using a GPU (Graphics Processing Unit).

Keywords: Sequence Alignment, A-Star, Sankoff Algorithm, GPU

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

BAliBASE Benchmark Alignment Database. xvii, 84, 87, 108, 110, 116, 121, 122, 153

BLOSUM Blocks Substitution Matrix. 10

CUDA Compute Unified Device Architecture. xv, xvii, 6, 60–68, 70, 76–79, 83, 95, 148

DU Dispatch Unit. 66, 67

FPGA Field Programmable Gate Arrays. 3, 4, 53, 58, 74, 75, 80, 81, 86, 92, 93, 98

GFLOP Giga Floating-point Operations Per Second. 70, 71

GPC Graphics Processing Cluster. 66–68

GPGPU General-Purpose Computing on Graphics Processing Units. 60

GPU Graphics Processing Unit. vii, viii, xviii, 3–6, 53, 60–62, 64, 65, 68, 70, 75–79, 82, 83, 85–88, 93–98, 121, 140, 141, 145, 147–150, 153, 154

HDL Hardware Description Language. 58

HT Hyper-Threading. 53, 113

LD/ST Load/Store. 64, 66, 67

MIC Many-Integrated Core. 59

MIMD Multiple Instruction Multiple Data. 52

MPI Message Passing Interface. 57, 59, 80, 88, 119, 121

MSA Multiple Sequence Alignment. 87, 88, 121

OpenCL Open Computing Language. 58, 61

OpenMP Open Multi-Processing. 53, 55–57, 59, 76, 79, 148

PAM Percent Accepted Mutations. xvii, 10

PD Precisão Dupla. 70

PDe Protein Design. 88, 121

PF Pathfinding. 88, 121

PS Precisão Simples. 70

Pthreads POSIX Threads. 53, 54, 57, 59, 128

RC Rubik Cube. 88, 121

SFU Special Function Units. 64, 66, 67

SIMD Single Instruction Multiple Data. 52, 59, 60

SM Streaming Multiprocessor. xv, 64–70

SMP Symmetric Multiprocessing. 52, 53, 57, 154

SPu Sliding Puzzle. 88, 121

STL Standard Template Library. 105

VHDL VHSIC Hardware Description Language. 58, 93

VPU Vector Processing Unit. 59

WS Warp Scheduler. 66, 67

Capítulo 1

Introdução

A Bioinformática é uma área multidisciplinar de pesquisa que visa o desenvolvimento de ferramentas e algoritmos para auxiliar biólogos na análise de dados, com o objetivo de determinar a função e/ou estrutura das sequências biológicas, inferir informação sobre sua evolução nos organismos, entre outros [87]. Uma das operações mais básicas da Bioinformática é a comparação de sequências biológicas.

A comparação de sequências pode ser feita em pares (2 sequências) ou com múltiplas sequências (3 ou mais sequências). Para resolver um problema de comparação de sequências, existem (a) algoritmos exatos, que retornam sempre a melhor solução (solução ótima) e (b) algoritmos heurísticos, que retornam uma boa solução para o problema. O problema do alinhamento múltiplo de sequências foi provado NP-Completo [133] e, por essa razão, algoritmos heurísticos são geralmente usados.

Para as situações onde se deseja obter a solução ótima na comparação de múltiplas sequências, existe um algoritmo simples, baseado em programação dinâmica, que é uma generalização do algoritmo ótimo para obtenção do alinhamento par-a-par [87]. Porém, devido à sua complexidade computacional exponencial, esse algoritmo não é utilizado. Carrillo-Lipman [8] mostraram que para se obter o resultado ótimo não é necessário procurar em todo o espaço de busca. Foi então estabelecido um limite conhecido como *Carrillo-Lipman bound*, que garante que nós que possuem um escore muito ruim podem ser descartados.

Em outra abordagem, o espaço de busca em questão pode ser modelado como um grafo. Desta forma, obter o alinhamento ótimo é equivalente a encontrar o caminho de menor custo neste grafo. A forma de solução mais amplamente conhecida para este tipo de busca é chamada de A-Star (A*) [109]. O algoritmo A-Star avalia os nós através do custo de se chegar ao nó e de uma estimativa de se chegar do nó atual até o nó final. O A-Star foi amplamente utilizado para a solução do problema do alinhamento múltiplo de sequências [54, 63, 77, 107, 113, 118] e os resultados mais rápidos são obtidos com a

utilização de alguma variante dele [37, 89].

Apesar do alinhamento múltiplo de sequências fornecer informação bastante relevante para vários problemas importantes de Bioinformática, em muitos casos, ele não é suficiente. Isso ocorre porque o alinhamento múltiplo leva em conta somente a estrutura primária da sequência biológica (isto é, os nucleotídeos ou aminoácidos que compõem o DNA, RNA ou proteínas). Nestes casos, a similaridade entre as sequências não se encontra na sequência de nucleotídeos, mas sim na estrutura espacial que elas formam.

Diferente do DNA, os nucleotídeos de uma sequência de RNA podem formar pares de bases e dobrar-se em si. Sendo assim, as sequência de RNA formam uma estrutura complexa, que pode ser representada em um plano como uma estrutura secundária. Em geral, a funcionalidade biológica está ligada à estrutura formada [27].

Com a publicação do genoma humano em 2001 [69] foi revelado que as partes codificadoras de proteínas tomam apenas 1,2% do genoma. Além disso, 50% do genoma é composto por sequências repetidas e é interessante que o projeto ENCODE (*ENCyclopedia Of DNA Elements*) revelou que aproximadamente 80% do genoma humano contém alguma função bioquímica, principalmente em regiões que não codificam proteínas, e que aproximadamente 75% do genoma é transcrito em RNA [12]. Este projeto compilou a anotação de 10.000 pequenos ncRNAs (RNAs não-codificadores) e outros 10.000 ncRNAs longos (lncRNAs). Descobertas similares foram reportadas em ratos e outros eucariotos e foi descoberto que a complexidade dos organismos aumenta com a quantidade de RNA não-codificador de proteína [34, 80]. Em outras palavras, a maioria do genoma de grandes eucariotos, tais como os mamíferos e os vertebrados, é transcrito em RNA que não codifica proteínas [80]. Por exemplo, aproximadamente metade do transcritoma obtido do cromossomo 21 do ser humano é não codificador [9] e por muitos anos essas regiões não codificadoras foram consideradas como lixo genético. Porém, hoje são conhecidas diversas funções biológicas desempenhadas pelos ncRNAs [13, 14, 22, 28, 138].

Como em geral a função biológica está ligada à estrutura, é plausível inferir que durante o processo de evolução ocorram mudanças compensatórias. Isto é, dois nucleotídeos que formam um par de base (por exemplo, A-U) são alterados por outros que também o formam (por exemplo, C-G). Então, como o par de bases é mantido, a estrutura dificilmente é alterada e a funcionalidade é preservada. Nestes casos, ferramentas que detectam apenas alterações na estrutura primária podem não ser precisas o suficiente para detectar este tipo de situação. Portanto, as funções biológicas estão presentes na estrutura espacial do RNA e não na sequência. Atualmente, usa-se a estrutura secundária (2D) para simplificar a modelagem da estrutura espacial.

O alinhamento estrutural de sequências de RNA produz não somente um alinhamento de sequências, mas também uma estrutura secundária comum entre elas. Esse problema

pode ser resolvido através do alinhamento e dobramento simultâneo, cuja solução ótima é dada pelo algoritmo de Sankoff [111]. Este algoritmo possui uma alta complexidade, admitindo sequências de tamanho próximo, e considerando n o tamanho das sequências e k o número de sequências, o algoritmo possui complexidade de tempo $O(n^{3k})$ de tempo e $O(n^{2k})$ de memória [111]. Por isso, as estratégias baseadas no algoritmo de Sankoff frequentemente utilizam heurísticas para reduzir sua complexidade computacional.

O uso de heurísticas para reduzir a complexidade computacional é uma estratégia muito utilizada na literatura. Para o alinhamento primário múltiplo podemos citar métodos heurísticos progressivos (ClustalW [129]) e iterativos (DIALIGN [85]). Para as sequências de RNA, também encontramos os métodos heurísticos e podemos citar o Foldalign [38] e LocARNA [134]. Apesar de serem capazes de obter uma solução em tempo razoável, esses métodos heurísticos não garantem o resultado ótimo.

Para acelerar a obtenção de resultados, muitas vezes são utilizadas técnicas de paralelismo utilizando arquiteturas de alto desempenho para obtenção de resultados em tempo reduzido. Dentre essas arquiteturas, podemos citar os *clusters* de computadores [53], os *Field Programmable Gate Arrays* (FPGAs) [135] , os aceleradores Intel Xeon Phi [58] e as unidades de processamento gráfico (GPUs) [110].

1.1 Motivação

Conforme dito anteriormente, Carrillo-Lipman [8] demonstraram que não é necessário percorrer todo o espaço de busca para a obter o alinhamento múltiplo ótimo de sequências. O algoritmo de busca A-Star também tem sido aplicado para a obtenção do alinhamento múltiplo [77, 118], também reduzindo o espaço de busca na obtenção do alinhamento ótimo. Um dos desafios para se utilizar o A-Star em paralelo é a forma da divisão do espaço de busca entre as *threads*, pois o padrão de expansão de nodos do grafo é irregular e potencialmente um número enorme de nodos pode ser gerado, consumindo rapidamente a memória disponível. Dentre as abordagens estudadas [37, 59, 89, 120, 140], não se encontraram estratégias que abordem os problemas de divisão do espaço de busca entre *threads* e uso de memória adicional simultaneamente.

Para a análise de sequências de RNA, muitas vezes é necessário considerar também a estrutura secundária na comparação. O algoritmo de Sankoff [111] produz a solução ótima para o alinhamento secundário de sequências de RNA, porém sua complexidade exponencial fez com que estratégias heurísticas fossem investigadas. O Foldalign [38] foi o primeiro algoritmo que introduziu heurísticas no algoritmo de Sankoff para reduzir a sua complexidade computacional e obter resultados em tempo razoável. Porém, na sua

última versão antes da elaboração da presente Tese, o Foldalign não possuía uma versão paralela.

Para se analisar a estrutura secundária de uma única sequência de RNA em arquiteturas de alto desempenho, as propostas que existem na literatura utilizam FPGA [57], Intel Xeon Phi [103] ou GPU [71, 72, 108]. Porém todas essas propostas fazem a previsão da estrutura secundária para apenas uma sequência e não conhecemos nenhuma proposta baseada no algoritmo de Sankoff para a produção do alinhamento secundário de sequências de RNA para duas ou mais sequências.

A principal motivação desta Tese é evoluir o estado da arte para a obtenção do alinhamento. Para isso, os algoritmos considerados foram divididos em duas categorias: algoritmos para obtenção do alinhamento primário e do alinhamento secundário. O alinhamento primário leva em conta somente a sequência de caracteres enquanto o alinhamento secundário leva em conta a estrutura 2D. O alinhamento primário em pares (ótimo ou heurístico) e o alinhamento primário múltiplo heurístico são amplamente estudados na literatura e por isso decidimos limitar o escopo dessa Tese. No caso do alinhamento múltiplo primário ótimo, visamos diminuir o espaço de busca de sequências utilizando o algoritmo A-Star. O alinhamento secundário de sequências de RNA também possui uma alta complexidade computacional e até onde sabemos não existe método exato para a redução do espaço de busca como existe no alinhamento primário. Por isso, para o alinhamento secundário, a presente Tese tem seu foco no alinhamento em pares. Para esse problema, consideramos tanto as soluções heurísticas como as soluções ótimas. Para o alinhamento secundário ótimo de sequências, até onde sabemos, não existe na literatura abordagem em arquiteturas de alto desempenho para o algoritmo de Sankoff.

1.2 Objetivos da Tese

O principal objetivo da presente Tese é propor e avaliar algoritmos paralelos para a solução do problema do alinhamento de sequências biológicas. Serão investigados os problemas para o alinhamento primário e secundário de sequências. Devido à alta complexidade desses problemas, as abordagens necessitam de técnicas para reduzir o tempo de execução e para isso serão utilizadas arquiteturas de alto desempenho.

1.2.1 Objetivos Específicos

São objetivos específicos desta Tese:

- Desenvolver um algoritmo paralelo para o alinhamento primário ótimo múltiplo de sequências com base no algoritmo A-Star que faça a atribuição de carga de trabalho com base na localidade e que trate o problema de alto consumo de memória;
- Propor uma estratégia paralela para o alinhamento heurístico secundário de pares de sequências de RNA a ser incorporada na ferramenta Foldalign;
- Desenvolver, implementar e avaliar uma estratégia paralela para execução do algoritmo de Sankoff para o alinhamento secundário de pares de sequências de RNA, utilizando GPUs.

1.3 Contribuições

Como contribuições desta Tese, foram desenvolvidas três soluções paralelas para a solução de diferentes problemas do alinhamento de sequências biológicas:

- PA-Star [125, 126]: Solução paralela em arquiteturas *multicore* para o alinhamento primário ótimo múltiplo de sequências biológicas que utiliza o algoritmo A-Star para reduzir o espaço de busca e funções de *hash* sensíveis à localidade para distribuir o espaço de busca entre as *threads*. Adicionalmente, foi proposto um módulo de escrita de dados de/para disco, visando expandir a quantidade de memória disponível para a execução do algoritmo [125];
- Foldalign 2.5 [123]: Solução paralela em arquiteturas *multicore* para o alinhamento secundário heurístico em pares de sequências de RNA. O Foldalign [38] é uma solução que utiliza heurísticas para reduzir a complexidade do problema e resolver o alinhamento secundário de sequências em pares. Nesta nova versão, foi adicionada uma estratégia paralela para reduzir o tempo total de execução em arquiteturas *multicore* mantendo todas as funcionalidades das versões anteriores;
- CUDA-Sankoff [122]: Solução paralela em GPU para o alinhamento secundário ótimo em pares de sequências de RNA. Essa solução é baseada no algoritmo de Sankoff [111] e, devido à sua alta complexidade, utilizamos GPUs e uma estratégia *wavefront* multinível para a redução do tempo total de execução. Até onde sabemos, não existe outra solução em GPU para a execução do algoritmo de Sankoff.

1.4 Sumário da Tese

Esta Tese está dividida em três partes. A primeira parte, Fundamentação Teórica, apresenta os conceitos de alinhamentos, arquiteturas de alto desempenho e os principais

algoritmos utilizados durante a elaboração desta Tese. Esta parte consiste em seis capítulos. O Capítulo 2 apresenta o alinhamento primário de sequências biológicas e seus principais algoritmos. O Capítulo 3 apresenta o problema de obtenção da estrutura secundária de sequências de RNA e seus principais algoritmos. Uma visão geral de arquiteturas de alto desempenho é dada no Capítulo 4, com foco na arquitetura CUDA (*Compute Unified Device Architecture*), utilizada nesse trabalho. Os Capítulos 5 e 6 apresentam uma revisão do estado da arte em alinhamento primário e secundário, respectivamente, em arquiteturas de alto desempenho.

A segunda parte da tese, Contribuições, descreve as contribuições da presente Tese. Esta parte contém três capítulos. O Capítulo 7 apresenta o PA-Star, uma estratégia paralela para a solução do alinhamento primário ótimo múltiplo. O Capítulo 8 apresenta o Foldalign 2.5, uma nova versão do programa Foldalign que utiliza uma estratégia paralela para a obtenção do alinhamento secundário heurístico em pares. Finalmente, o Capítulo 9 apresenta o CUDA-Sankoff, uma versão baseada em GPU para a solução do alinhamento secundário ótimo em pares.

Na terceira parte desta tese, o Capítulo 10 apresenta a conclusão e os trabalhos futuros a serem desenvolvidos. Finalmente, o Anexo A inclui a produção científica e a primeira página dos artigos publicados na presente Tese, sendo dois artigos publicados em periódicos internacionais [123, 125] e dois artigos publicados em conferências internacionais [122, 126].

Parte I

Fundamentação Teórica e Revisão Bibliográfica

Capítulo 2

Alinhamento Primário de Sequências Biológicas

Um dos problemas mais básicos em Bioinformática é a comparação de sequências biológicas [87]. Esta operação possui como resultado um escore, que representa a similaridade entre as sequências e, opcionalmente, pode oferecer um alinhamento que representa claramente as semelhanças e diferenças entre as sequências.

Do ponto de vista matemático, a comparação entre sequências biológicas configura-se como um problema de reconhecimento aproximado de padrões [19]. Podem ser comparadas duas sequências (alinhamento em pares) ou mais de duas sequências (alinhamento múltiplo). O alinhamento primário é o alinhamento de sequências que leva em consideração apenas as sequências, mas não leva em consideração a estrutura 2D ou 3D delas. Neste capítulo será detalhado o problema do alinhamento primário de sequências biológicas.

2.1 Alinhamento de Sequências Biológicas

As sequências biológicas são consideradas cadeias ordenadas de DNA, RNA ou proteínas.

Sequências de DNA e RNA são compostas por nucleotídeos, representados pelo alfabeto $\Sigma_{\text{DNA}} = \{A, T, G, C\}$ e $\Sigma_{\text{RNA}} = \{A, U, G, C\}$, respectivamente. As proteínas são sequências de aminoácidos do alfabeto $\Sigma_{\text{proteínas}} = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$.

Uma sequência de n caracteres é definida como um subconjunto do produto de caracteres de um alfabeto α de m caracteres, $\alpha = \{\alpha_1, ..., \alpha_m\}$ [87]:

$$S^n = \prod_{i=1}^{n} \alpha_i$$

Considere também o alfabeto β obtido do alfabeto α adicionando-se um espaço em branco "$-$" (*gap*):

$$\beta = \{-\} \cup \{\alpha_1, ..., \alpha_m\}$$

Em β podemos definir um alinhamento das k sequências $S_1, ..., S_k$, formado por um outro conjunto, $\bar{S}_1, ..., \bar{S}_k$, de tal forma que cada sequência $\bar{S}_i$ é obtida a partir de S_i, inserindo-se *gaps* em posições onde alguma das outras sequências possui um caractere que não é *gap* [19].

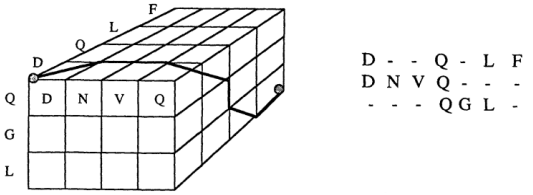


Figura 2.1: Alinhamento de três sequências [8].

Em cada sequência, os caracteres das sequências podem ser iguais (*match*) ou diferentes (*mismatch*). Cada alinhamento de sequências codifica um diferente conjunto de inserções, remoções e substituições, de maneira que as semelhanças e diferenças sejam destacadas. A Figura 2.1 representa um alinhamento das sequências de aminoácidos $S_1 = DQLF, S_2 = DNVQ$, e $S_3 = QGL$. Cada coluna do alinhamento de sequências possui um um escore, e a soma de todos compõe o escore total, que é o valor que quantifica a similaridade do alinhamento [87].

2.1.1 Características dos Alinhamentos de Sequências

Para o problema de alinhamento entre sequências biológicas é necessário definir diversas características para saber qual tipo de alinhamento está sendo obtido. Inicialmente temos que definir o tipo de sequência que será considerada. As sequências podem variar, podendo ser de bases nitrogenadas como DNA ou RNA, ou podem ser de aminoácidos, que formam sequências de proteínas (Seção 2.1).

Em linhas gerais, o alinhamento de sequências pode ser classificado como global ou como local. No alinhamento global de sequências, todos os caracteres das sequências são incluídos no alinhamento. No alinhamento local, o alinhamento é feito por subsequências, desta forma são descartados prefixos e sufixos que não sejam muito semelhantes [87].

Dado um conjunto de sequências, existem diversos alinhamentos possíveis entre elas. Consideramos que um alinhamento é ótimo quando ele é um dos alinhamentos que possuem o melhor escore dentre todos os alinhamentos possíveis. Algoritmos que obtém um

alinhamento, porém não garantem que é o melhor escore possível, são chamados algoritmos heurísticos.

Outra característica importante é o sistema de escore utilizado. A função que obtém o escore pode associar um valor único a *matches*, *mismatches* e *gaps*, ou pode utilizar matrizes de substituição. As matrizes de substituição são matrizes 4x4 ou 20x20 que contêm o escore para cada par possível de bases/aminoácidos (*matches* ou *mismatches*). Esses escores são obtidos com dados biológicos estatisticamente relevantes. As matrizes de substituição mais amplamente conhecidas são a PAM [15] e a BLOSUM [45]. A Tabela 2.1 ilustra a matriz PAM 250.

Tabela 2.1: Matriz de substituição PAM 250 [15].

	-	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
-	0	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
C	12	05	17	19	20	19	20	21	22	22	22	20	21	22	22	19	23	19	21	17	25
S	12	17	15	16	16	16	16	16	17	17	18	18	17	17	19	18	20	18	20	20	19
T	12	19	16	14	17	16	17	17	17	17	18	18	18	17	18	17	19	17	20	20	22
P	12	20	16	17	11	16	18	18	18	18	17	17	17	18	19	19	20	18	22	22	23
A	12	19	16	16	16	15	16	17	17	17	17	18	19	18	18	18	19	17	21	20	23
G	12	20	16	17	18	16	12	17	16	17	18	19	20	19	20	20	21	18	22	22	24
N	12	21	16	17	18	17	17	15	15	16	16	15	17	16	19	19	20	19	21	19	21
D	12	22	17	17	18	17	16	15	13	14	15	16	18	17	20	19	21	19	23	21	24
E	12	22	17	17	18	17	17	16	14	13	15	16	18	17	19	19	20	19	22	21	24
Q	12	22	18	18	17	17	18	16	15	15	13	14	16	16	18	19	19	19	22	21	22
H	12	20	18	18	17	18	19	15	16	16	14	11	15	17	19	19	19	19	19	17	20
R	12	21	17	18	17	19	20	17	18	18	16	15	11	14	17	19	20	19	21	21	15
K	12	22	17	17	18	18	19	16	17	17	16	17	14	12	17	19	20	19	22	21	20
M	12	22	19	18	19	18	20	19	20	19	18	19	17	17	11	15	13	15	17	19	21
I	12	19	18	17	19	18	20	19	19	19	19	19	19	19	15	12	15	13	16	18	22
L	12	23	20	19	20	19	21	20	21	20	19	19	20	20	13	15	11	15	15	18	19
V	12	19	18	17	18	17	18	19	19	19	19	19	19	19	15	13	15	13	18	19	23
F	12	21	20	20	22	21	22	21	23	22	22	19	21	22	17	16	15	18	8	10	17
Y	12	17	20	20	22	20	22	19	21	21	21	17	21	21	19	18	18	19	10	07	17
W	12	25	19	22	23	23	24	21	24	24	22	20	15	20	21	22	19	23	17	17	00

2.2 Comparação em Pares

O problema do alinhamento em pares consiste em identificar o grau para similaridade entre duas sequências através de um escore de semelhança e, idealmente, visualizar as semelhanças e diferenças entre as sequências utilizando um alinhamento.

Na presente Tese, estamos interessados em obter o alinhamento ótimo de sequências biológicas. No restante desta seção serão descritos algoritmos para obter os alinhamentos ótimos local e global.

2.2.1 Algoritmo de Needleman-Wunsch (NW)

O algoritmo Needleman-Wunsch [88] resolve o problema do alinhamento ótimo global entre duas sequências S_1 e S_2 de comprimento n_1 e n_2.

Para isto, este algoritmo calcula uma matriz D, onde o elemento $D_{i,j}$ representa o escore ótimo dos prefixos das sequências $S_1[1..i]$ e $S_2[1..j]$. O cálculo desta matriz utiliza a técnica de programação dinâmica [16]. Considere que a função $cost(a, b)$ retorna o custo de *match* ou *mismatch* dos caracteres "a" e "b". Considere também a penalidade de *gaps* $sspace$. O primeiro passo do algoritmo é iniciar a primeira célula da matriz $D_{0,0} = 0$. Em seguida a primeira linha e a primeira coluna são iniciadas com valores da equação 2.1:

$$\begin{aligned} D_{i,0} &= D_{i-1,0} - sspace \\ D_{0,j} &= D_{0,j-1} - sspace \\ \forall (i,j) &: 0 < i \leq n_1, 0 < j \leq n_2. \end{aligned} \tag{2.1}$$

Para as outras células da matriz é aplicada a fórmula de recorrência apresentada na Equação 2.2:

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + cost(S_1[i], S_2[j]) \\ D_{i-1,j} - sspace \\ D_{i,j-1} - sspace \end{cases} \tag{2.2}$$

A fórmula de recorrência modela três casos. O primeiro é o caso onde o melhor escore da célula $D_{i,j}$ seria o alinhamento dos prefixos $S[0..i-1], S_2[0..j-1]$ acrescido dos caracteres $S_1[i]$ e $S_2[j]$. Os outros dois casos modelam a inserção de um *gap* em uma das sequências.

A fase de inicialização das células e aplicação da equação de recorrência é a primeira fase do algoritmo. Nela, enquanto a equação de recorrência é aplicada, o algoritmo guarda um valor de referência indicando qual célula foi utilizada. Esta fase termina quando o valor da célula $D_{i,j}$ é calculado.

Ao término da primeira fase, o maior escore está presente na célula D_{n_1,n_2} e representa o escore ótimo (valor de similaridade) entre as sequências, porém para visualizar as diferenças e semelhanças entre as sequências, também é desejável obter o alinhamento.

O alinhamento é obtido na segunda fase do algoritmo, chamada fase de *traceback*. Iniciando na célula D_{n_1,n_2}, deve-se verificar que célula foi utilizada para o cálculo. Dependendo da célula, decide-se então se deve ser adicionado um *gap* em uma das sequências, ou se o valor representa um *match* ou *mismatch*. Esta etapa é repetida até que se atinja a célula inicial, $D_{0,0}$. A Figura 2.2 ilustra uma matriz de programação dinâmica H e as células que foram percorridas durante o *traceback* estão destacadas na cor cinza. Os valores para *match*, *mistach* e *gap* são, respectivamente +1, -1 e -2.

É fácil observar que para preencher a matriz D o algoritmo necessita de $n_1 \times n_2$ operações. Considerando que normalmente o tamanho das sequências é próximo, considera-se

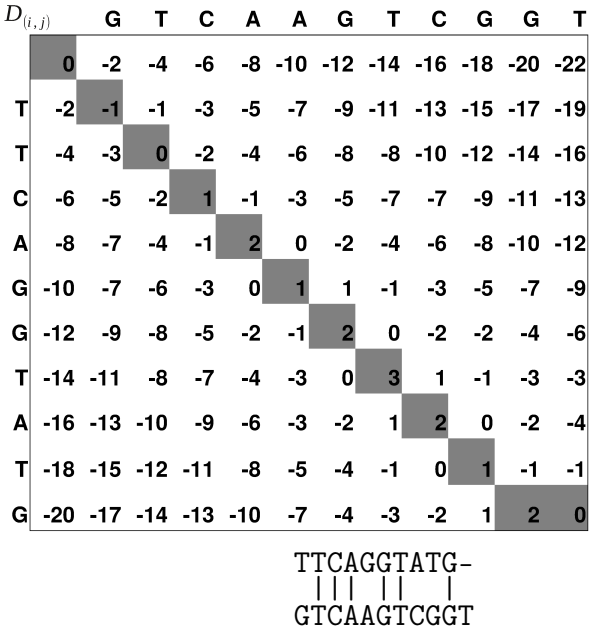


Figura 2.2: Matriz de programação dinâmica do algoritmo Needleman-Wunsch para as sequências $GTCAAGTCGGT$ e $TCCAAGTATG$. A região em cinza mostra as células percorridas durante a fase de *traceback*. O alinhamento produzido pelo algoritmo é mostrado abaixo da matriz.

que a complexidade de tempo e memória do algoritmo é $O(n^2)$. Já na fase de *traceback*, o limite superior é da ordem de $O(n_1 + n_2)$ operações.

2.2.2 O Algoritmo de Smith-Waterman (SW)

O algoritmo de Smith-Waterman [116] resolve o problema do alinhamento ótimo local em pares de sequências biológicas. Esse tipo de alinhamento é desejável para alinhar subsequências de S_1 e S_2.

As fases do algoritmo Smith-Waterman são bem similares ao algoritmo de Needleman-Wunsch (Seção 2.2.1). A equação de recorrência é alterada para impedir valores negativos. Isso quer dizer que quando um valor negativo é encontrado, é melhor descartar o alinhamento obtido até então, pois a similaridade entre essas porções das sequências está baixa, sendo melhor reiniciar o alinhamento a partir de uma nova posição. A equação de recorrência está descrita na Equação 2.3.

$$D_{i,j} = \max \begin{cases} 0 \\ D_{i-1,j-1} + cost(S_1[i], S_2[j]) \\ D_{i-1,j} - sspace \\ D_{i,j-1} - sspace \end{cases} \tag{2.3}$$

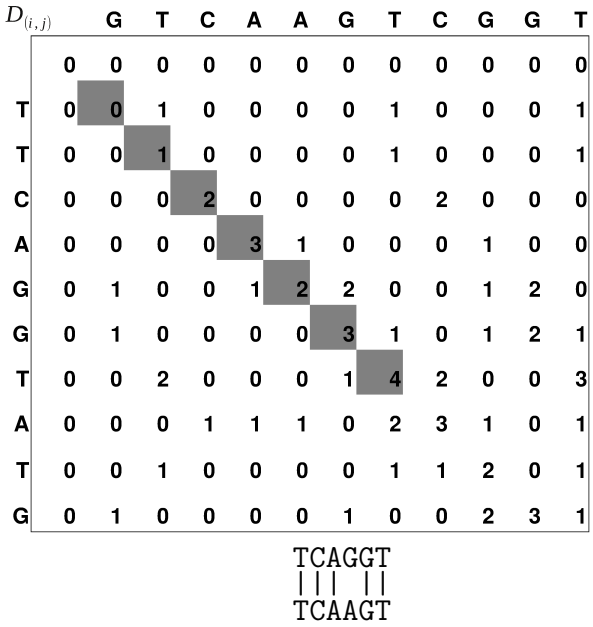


Figura 2.3: Matriz de programação dinâmica do algoritmo Smith-Waterman para as sequências $GTCAAGTCGGT$ e $TCCAAGTATG$. A região em cinza mostra as células percorridas durante a fase de *traceback*. O alinhamento produzido pelo algoritmo é mostrado abaixo da matriz.

Com isto, não apenas a equação de recorrência é alterada, mas também o preenchimento da primeira linha e coluna. Enquanto no algoritmo de Needleman-Wunsch essas primeiras linha e coluna são preenchidas com valores múltiplos negativos da penalidade de *gap*, no algoritmo de Smith-Waterman são preenchidas com apenas zero, conforme a Equação 2.4:

$$\begin{aligned} D_{i,0} &= 0 \\ D_{0,j} &= 0 \\ \forall (i,j) : 0 < i \leq n_1, 0 < j \leq n_2. \end{aligned} \tag{2.4}$$

A terceira mudança está na fase de *traceback*. O algoritmo de Needleman-Wunsch sempre inicia o *traceback* na posição D_{n_1,n_2}. No algoritmo de Smith-Waterman, é necessário salvar em uma variável à parte a posição em que se encontra o melhor valor de toda a matriz. Então o *traceback* é iniciado desta posição até encontrar um valor igual a zero. Desta forma encontra-se o melhor alinhamento possível desconsiderando-se prefixos e sufixos que não contribuam significativamente para o escore do alinhamento. Um exemplo de matriz de programação dinâmica de Smith-Waterman é apresentado na Figura 2.3, com valores de *match*, *mismatch* e *gap* sendo, respectivamente, 1, -1 e -2.

2.3 Comparação Múltipla de Sequências Biológicas

O alinhamento em pares de sequências representa as diferenças e semelhanças entre duas sequências. Porém muitas vezes é desejável obter o alinhamento de um conjunto de sequências, e não apenas de um par específico.

Neste caso, utiliza-se um algoritmo que faça a comparação de não apenas duas, mas de k sequências biológicas, tal que $k > 2$ é um inteiro [32]. Este problema é conhecido como alinhamento múltiplo de sequências, e ele produz um alinhamento onde mais de duas sequências são comparadas simultaneamente. Desta forma, as diferenças e semelhanças entre um conjunto de sequências podem se tornar mais claras [87]. Assim como o alinhamento em pares, o alinhamento múltiplo ótimo é definido como um dos alinhamentos que possui o melhor escore, dentre todos os possíveis alinhamentos. O escore de um alinhamento múltiplo é geralmente dado pela soma dos escores de todos os alinhamentos par-a-par, conhecido como *sum-of-pairs* [32]. O problema de obtenção do alinhamento múltiplo com escore *sum-of-pairs* foi provado NP-Completo [133] e, devido à complexidade computacional alta, geralmente heurísticas são aplicadas a este problema.

Pode-se agrupar os métodos heurísticos em sub-grupos de acordo com a estratégia utilizada [32]. Os métodos heurísticos progressivos normalmente alinham as sequências mais similares e depois adicionam as outras sequências no alinhamento múltiplo. Os métodos heurísticos iterativos também buscam adicionar outras sequências a um alinhamento já existente, porém estabelecem que alguns subgrupos de sequências podem ser revisados a fim de minimizar e/ou corrigir os erros do alinhamento inicial.

Nesta seção descrevemos o alinhamento primário global múltiplo de sequências biológicas, descrevendo inicialmente as soluções ótimas e em seguida as soluções heurísticas.

2.3.1 Comparação Múltipla Ótima

O problema do alinhamento global múltiplo de sequências consiste em computar um alinhamento global múltiplo M com um escore ótimo [87]. Apesar de utilizarmos o escore mínimo neste trabalho, o escore máximo, como no caso de Smith-Waterman [116] e Needleman-Wunsch [88], também pode ser utilizado, e a conversão entre as equações de recorrência é trivial. A seguir, será apresentado o caso de três sequências, que pode ser generalizado para um número qualquer de sequências.

Considere S_1, S_2 e S_3, três sequências de tamanho n_1, n_2 e n_3, respectivamente, e seja $D(i, j, k)$ o escore ótimo soma dos pares (*sum-of-pairs*) para o alinhamento de $S_1[1..i], S_2[1..j]$ e $S_3[1..k]$. Considere também o escores para *match*, *mismatch* ou *gap* como sendo *smatch*, *smis* e *sspace*, respectivamente.

Para obter o escore ótimo do alinhamento múltiplo de três sequências, um cubo é utilizado e cada célula (i, j, k) que não está na fronteira (ou seja, todos os índices são diferentes de zero) possui sete vizinhos que precisam ser consultados para determinar $D(i, j, k)$.

Para computar os valores das fronteiras, considere $D'_{1,2}(i, j)$ como sendo o valor do escore obtido pela soma de pares subsequências $S_1[1..i]$ e $S_2[1..j]$. Definindo $D'_{1,3}(i, k)$ e $D'_{2,3}(j, k)$ por analogia, temos [32]:

$$
\begin{aligned}
D(i, j, 0) &= D'_{1,2}(i, j) + (i + j) * sspace; \\
D(i, 0, k) &= D'_{1,3}(i, k) + (i + k) * sspace; \\
D(0, j, k) &= D'_{2,3}(j, k) + (j + k) * sspace; \\
D(0, 0, 0) &= 0
\end{aligned}
$$

O Algoritmo 1 apresenta o algoritmo exato *naïve* para o alinhamento de 3 sequências. Neste algoritmo, admite-se que a linha 0 e a coluna 0 são inicializadas com 0. Existem 3 laços "for", (linhas 1 a 3), indicando cada uma das três sequências. Inicialmente, são calculados os escores para *matches* e *mismatches* (linhas 4 a 6). Depois disso, 7 valores são calculados (linhas 7 a 13), que representam as 7 células vizinhas que devem ser consultadas. Após isso, $D(i, j, k)$ recebe o menor desses valores (linha 14).

Algoritmo 1 : `Alinhamento Múltiplo de Sequências Naïve`

```
1:  for i = 1 → n1 do
2:      for j = 1 → n2 do
3:          for k = 1 → n3 do
4:              c_ij = CostMatchMismatch(S1(i), S2(j));
5:              c_ik = CostMatchMismatch(S1(i), S3(k));
6:              c_jk = CostMatchMismatch(S2(k), S3(j));
7:              d1 = D(i − 1, j − 1, k − 1) + c_ij + c_ik + c_jk
8:              d2 = D(i − 1, j − 1, k) + c_ij + gap
9:              d3 = D(i − 1, j, k − 1) + c_ik + gap
10:             d4 = D(i, j − 1, k − 1) + c_jk + gap
11:             d5 = D(i − 1, j, k) + 2 * gap
12:             d6 = D(i, j − 1, k) + 2 * gap
13:             d7 = D(i, j, k − 1) + 2 * gap
14:             D(i, j, k) = Min[d1, d2, d3, d4, d5, d6, d7]
15:         end for
16:     end for
17: end for
```

O problema do alinhamento múltiplo de sequências foi resolvido neste caso utilizando programação dinâmica. No caso geral de alinhamento múltiplo de sequências, considerando k sequências de tamanho n, constrói-se uma matriz n-dimensional, assim o algoritmo exige tempo $\Theta(n^k)$ [32].

2.3.1.1 Carrillo-Lipman (CL)

Como visto na seção 2.3.1, o espaço de busca na comparação múltipla de sequências é proporcional ao tamanho das sequências e exponencial em relação ao número delas. Porém, algumas células desse espaço de busca não precisam ser calculadas e mesmo assim a otimalidade do algoritmo pode ser garantida. Apesar de não implementar ou descrever um algoritmo, o método Carrillo-Lipman [8] foi o primeiro proposto para reduzir o espaço de busca para o problema do alinhamento múltiplo das sequências.

Baseado nos escores de um alinhamento heurístico e no alinhamento exato dos pares de sequências, um limite é encontrado e células que possuam um valor maior que este limite não podem fazer parte de um alinhamento ótimo, conforme a Figura 2.4.

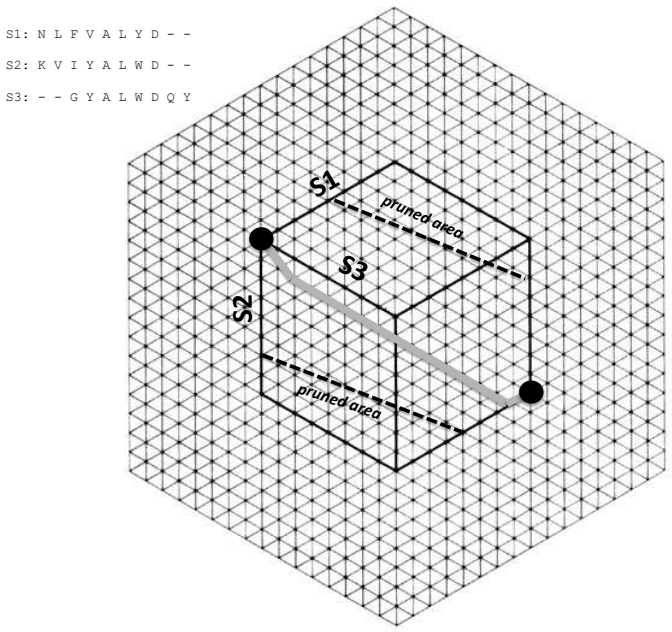


Figura 2.4: Alinhamento múltiplo com 3 sequências. O alinhamento múltiplo final é mostrado no canto superior esquerdo e a linha dentro do cubo ilustra o alinhamento dentro da matriz 3D de programação dinâmica [125].

Neste caso, o caminho do alinhamento ótimo passa por alguma outra parte do espaço de busca e o valor da célula descartada não é relevante para o cálculo do escore do alinhamento múltiplo ótimo. A seguir, são mostradas algumas observações demonstradas por Carrillo e Lipman [8] sobre o problema de se obter o alinhamento ótimo $\gamma^*(S_1, ..., S_n)$ onde $n > 2$.

Considere μ_{ij} uma medida de escore de duas sequências S_i e S_j. Nesse caso, a medida M de um caminho N-dimensional γ é:

$$M(\gamma) = \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma)) \tag{2.5}$$

Onde p_{ij} é a projeção em γ do plano determinado pelas sequências S_i e S_j e $M(\gamma)$ é a medida de escore deste alinhamento.

Considere agora o alinhamento N-dimensional do caminho γ^e, que será chamado γ-estimado. Este é algum possível alinhamento das sequências, que não é necessariamente o ótimo. Por isso, temos as Inequações 2.6 e 2.7:

$$M(\gamma^e) - M(\gamma^*) \geq 0 \tag{2.6}$$

$$\sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^*)) \geq 0 \tag{2.7}$$

Denota-se γ^*_{ij} como sendo um caminho ótimo para a medida μ_{ij} das sequências S_i e S_j. Como $\mu_{ij}(p_{ij}(\gamma^*)) \leq \mu_{ij}(\gamma^*_{ij})$, temos a Fórmula 2.8:

$$\sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{i<j,(i,j)\neq(k,l)}^{N} \mu_{ij}(p_{ij}(\gamma^*)) \geq \mu_{kl}(p_{kl}(\gamma^*)) \tag{2.8}$$

Assim define-se U_{kl} conforme a Equação 2.9:

$$U_{kl} = \sum_{i<j}^{N} \mu_{ij}(p_{ij}(\gamma^e)) - \sum_{i<j,(i,j)\neq(k,l)}^{N} \mu_{ij}(p_{ij}(\gamma^*)) \tag{2.9}$$

Desta forma, U_{kl} é um limite superior para a medida das projeções de qualquer caminho ótimo N-dimensional, feitas no plano determinado pelas sequências S_k e S_j. Então, quando procuramos pelo alinhamento ótimo γ^*, apenas precisamos considerar os caminhos γ em $L(S_1, ..., S_n)$ que satisfazem $\mu_{kl}(p_{kl}(\gamma)) \leq U_{kl}$.

Define-se então o conjunto P como sendo o conjunto composto pelos caminhos que satisfazem essa propriedade (Equação 2.10) :

$$P = \bigcap_{i<j}^{N} P_{ij} \tag{2.10}$$

Desta forma, para a obtenção do alinhamento ótimo, não é necessário aplicar o algoritmo de programação dinâmica em todo o espaço de busca $L(S_1, ..., S_n)$, mas é suficiente procurar na sub-região P.

Em trabalhos posteriores [31, 79], as equações que definem o Carrillo-Lipman *bound* são mostradas em forma simplificada, com uma notação diferente:

Define-se L como sendo um limite inferior da soma de pares [31] (Equação 2.11) :

$$L = \sum_{i<j} d(S_i, S_j) \cdot scale(S_i, S_j) \tag{2.11}$$

Onde $scale(S_i, S_j)$ é o peso atribuído às sequências S_i e S_j, $d(S_i, S_j)$ é o escore do alinhamento ótimo em par das sequências e L é o limite inferior definido, pois como o alinhamento em pares é ótimo, logo não existe um outro alinhamento com menor escore das sequências S_i e S_j. O escore do alinhamento múltiplo ótimo de todas as sequências é uma soma de pares, assim a soma dos pares ótimos das sequências deve ser menor ou igual ao escore do alinhamento múltiplo ótimo.

Considere um limite superior U, A^o o alinhamento ótimo, $c(A^o)$ como o escore do alinhamento múltiplo ótimo e $c(A^o_{i,j})$ o escore do alinhamento dos pares de sequências i e j no alinhamento A^o. Logo, temos a Inequação 2.12:

$$U - L \geq \sum_{i<j} [scale(S_i, S_j) \cdot (c(A^o_{i,j}) - d(S_i, S_j))] \tag{2.12}$$

Então, para quaisquer pares de sequências p e q, a Inequação 2.13 é válida:

$$U - L \geq scale(S_i, S_j) \cdot (c(A^o_{p,q}) - d(S_p, S_q)) \tag{2.13}$$

Rearranjando a inequação, obtemos a o Carrillo-Lipman *bound* [31] [79] (Inequação 2.14):

$$scale(S_i, S_j) \cdot c(A^o_{i,j}) \leq scale(S_i, S_j) \cdot d(S_i, S_j) + U - L \tag{2.14}$$

Considere um alinhamento heurístico A^h. Este alinhamento induz uma soma de escore c na soma de pares i e j, assim $U - L$ pode ser obtido pela Equação 2.15:

$$U - L = \sum_{i<j} [c(A^h_{i,j}) - d(S_i, S_j)] \tag{2.15}$$

A Equação 2.15 mostra que os limites superior e inferior possuem um valor baseado na projeção de um alinhamento heurístico subtraído dos custos dos alinhamentos em pares das sequências, garantindo assim um limite para que as células contribuam para encontrar o alinhamento ótimo.

2.3.1.2 MSA 2.0

D. Lipman e S. Altschul implementaram o MSA 1.0 [74] em 1989, que é o primeiro programa que utiliza o Carrillo–Lipman *bound* para a redução do espaço de busca para a obtenção do alinhamento ótimo.

O programa foi implementado utilizando a linguagem de programação C e foi testado em computadores com sistema operacional UNIX-like. Os requisitos de memória e processamento são uma função do número de sequências, do seu comprimento e do tamanho dos limites superiores para os alinhamentos em pares. Na época de sua implementação, foi possível alinhar cinco sequências de diferentes famílias da superfamília das globinas utilizando um tempo inferior a dois minutos, sendo necessário menos de 1,3 Megabytes de memória.

Gupta *et al.* [31] propuseram em 1995 uma versão posterior onde melhoraram os requisitos de memória e tempo de execução do programa MSA 1.0, criando a versão MSA 2.0.

Por padrão, o MSA 2.0 utiliza *weighted sum-of-pairs* e utiliza um método de árvore filogenética para calcular o peso das sequências [1].

A economia de recursos é feita pelo MSA 2.0 pois ele procura apenas entre alinhamentos que possuem um custo $\leq U$. A parte principal do MSA 2.0 considera que um valor finito U é conhecido. Se U for muito pequeno, o programa não é capaz de encontrar alguma solução. Por esta razão, podem ser necessárias múltiplas execuções do programa incrementando o valor de U para se obter um alinhamento, ótimo ou não. O motivo de não se utilizar um valor U arbitrário e muito grande é que este valor é diretamente proporcional à quantidade de memória e processamento exigidos pelo programa, e pode fazer com que a a finalização do programa demore muito.

O valor $U - L$ (Equação 2.15) é armazenado em uma variável δ. Existem três formas para se calcular δ e em todas elas U é calculado como $L + \delta$. Na primeira forma, o usuário especifica δ. Na segunda forma, o programa calcula um alinhamento múltiplo heurístico que não requer muita memória nem processamento. O alinhamento heurístico produz um custo de alinhamento em pares. O valor $\epsilon_{i,j}$ é a diferença entre o custo deste alinhamento e o custo do alinhamento ótimo em pares $d(S_i, S_j)$. Sendo assim δ é calculado conforme a Equação 2.16:

$$\delta = \sum_{i<j} (min(\textbf{maxepsilon}, \epsilon_{i,j}).scale(S_i, S_j). \tag{2.16}$$

O número **maxepsilon** é 50 por padrão, mas pode ser modificado em uma constante no código. A terceira forma é que o usuário pode passar para o programa um arquivo com valores para $\epsilon_{i,j}$, neste caso os valores são usados da mesma forma que os valores

ϵ computados na segunda opção. Este método pode levar a valor U muito baixo. O algoritmo MSA 2.0 é organizado em duas fases. Na primeira, $O(n^2)$ comparações em pares são feitas para construir o conjunto $F_{i,j}$. Na segunda fase, um alinhamento que possui o menor escore é computado onde é garantido que as projeções do caminho ótimo estejam todas dentro de $F_{i,j}$.

Ao final, a função $TRACEBACK(t)$ retorna um alinhamento ótimo possível, percorrendo o caminho inverso, no sentido final até o início, examinando a distância e os pontos dos vértices predecessores.

Gupta *et al.* [31] compararam o desempenho do MSA 1.0 com o MSA 2.0. Os programas foram testados em uma estação Sun Sparcstation 10 com 128 MB de memória RAM e compilados com flag de otimização. Os resultados da comparação do desempenho mostram que, para todos os casos, houve redução no tempo de execução e consumo de memória, chegando a obter um tempo de execução menor em $5,7$ vezes e um consumo de memória menor $13,9$ vezes.

Os alinhamentos produzidos pelo MSA 2.0 foram comparados com os resultados em McClure *et al.* [82] proposto em 1994. Nessa comparação, cada *motif* é um bloco de uma a cinco colunas consecutivas alinhadas, obtidas manualmente por um especialista. Para cada conjunto de sequências, é reportada a soma dos *motifs* alinhados corretamente. Para a Globins A, que possui 5 *motifs*, 4 deles foram alinhados corretamente pelo MSA 2.0 e, no quinto *motif*, 6 de 7 sequências foram alinhadas corretamente. Logo, o MSA 2.0 produziu um resultado que não era o desejado pelos biólogos, mas por se tratar de um alinhamento ótimo, sabemos que o erro não pode estar no alinhamento produzido. Isso se deve ao fato do alinhamento múltiplo ótimo ser definido como o melhor matematicamente, porém devido à simplificações no modelo, às vezes o melhor alinhamento obtido não corresponde a realidade do alinhamento na natureza.

2.3.2 Comparação Múltipla Ótima com o Algoritmo A-Star

O problema do alinhamento múltiplo de sequências pode ser descrito como um grafo e, neste caso, o problema do alinhamento ótimo é equivalente a encontrar o menor caminho entre o nó inicial de coordenadas $C_i = (0, 0, ..., 0)$ e o nó final de coordenadas $C_f = (n_1, n_2, ..., n_n)$, onde n_i é o comprimento da sequência i e k é o número de sequências. Hart *et al.* [35, 36] propuseram um dos algoritmos de busca mais amplamente conhecidos: A-Star (A*, pronuncia-se A-Estrela). Para reduzir o espaço de busca, uma heurística admissível é utilizada. No algoritmo A-Star, a avaliação dos nós é dada pela Equação 2.17. Esta equação descreve a ordem que os nós do grafo devem ser visitados durante a

busca pelo resultado ótimo.

$$f(N) = g(N) + h(N) \tag{2.17}$$

Na equação 2.17, $g(N)$ representa o custo para se alcançar um nó, $h(N)$ é custo estimado do caminho de menor custo de N até o objetivo e $f(N)$ representa o custo estimado da solução de menor custo através de N. Neste algoritmo, a estrutura coordenada armazena uma posição única no grafo a ser percorrido e a estrutura nó armazena os valores f, g, h e a coordenada.

A função $h(N)$ utiliza informações do ambiente adicionais para fazer tal estimativa e varia de acordo com o tipo de problema que o algoritmo está sendo aplicado [109]. Por exemplo, dado um grafo G, onde cada nó N deste grafo representa uma cidade, T_i é a cidade inicial, T_f representa a cidade objetivo, podemos utilizar o A-Star para encontrar o menor caminho entre as cidades T_i e T_f. Neste caso, $g(N)$ representa a distância real a ser percorrida entre alguma cidade T_x, e a cidade inicial T_i. Para estimar a distância entre a cidade T_x e a cidade destino T_f (ou um conjunto de cidades destino), frequentemente é utilizada a heurística **distância em linha reta**. Afinal, a distância em linha reta representa uma estimativa, sempre inferior ou igual à distância real. A combinação destes valores produz $f(n)$, o valor estimado da solução de menor custo através de T_x.

2.3.2.1 Algoritmo A-Star

Algoritmo 2 *A-Star(C_i, C_f)*

```
1: N_i.c ← C_i /* Cria um nó inicial N_i de coordenadas C_i */
2: N_i.g = 0
3: N_i.h = get_h(C_i) /* Estimativa de custo do nó inicial */
4: N_i.f = N_i.g + N_i.h
5: OpenList ← N_i /* Adiciona o nó inicial à OpenList */
6: while OpenList.lowest_f() ∉ C_f and OpenList ≠ ∅ /* Verifica condição de parada */ do
7:     /* Início do passo de busca */
8:     current ← OpenList.get_lowest_f() /* Remove nó de melhor prioridade da OpenList */
9:     if current.g ≤ ClosedList.find_g(current) then
10:        ClosedList ← ClosedList ∪ current /* Adiciona à ClosedList */
11:        for neighbor in neigh(current) /* Fase de expansão */ do
12:            if neighbor.g < OpenList.find_g(neighbor) and neighbor.g < ClosedList.find_g(neighbor) then
13:                OpenList ← OpenList ∪ neighbor /* Adiciona os vizinhos à OpenList */
14:            end if
15:        end for
16:    end if
17: end while
18: if OpenList = ∅ then
19:     n_e ← NULL
20: else
21:     n_e ← OpenList.get_lowest_f()
22: end if
23: return n_e
```

O Algoritmo A-Star é apresentado no Algoritmo 2. O Algoritmo recebe como parâmetros a coordenada inicial e a final, C_i e C_f, respectivamente. A *OpenList* contém os

nós que devem ser expandidos e a *ClosedList* contém os nós que já foram expandidos. Inicialmente, um nó inicial é criado (linhas 1-4) a partir da coordenada inicial C_i, com custo g igual a zero e estimativa h obtida através de uma função auxiliar get_h. Esta estimativa varia de acordo com o tipo de busca que está sendo realizada (distância entre cidades, alinhamento múltiplo, etc...). Nesta etapa, a *ClosedList* está vazia e a *OpenList* contém apenas o nó inicial N_i. A seguir, se o nó selecionado pertencer ao conjunto de nós com a coordenada final C_f, ou se o não existir nó na *OpenList*, o algoritmo para (linha 6). Senão, o nó de melhor prioridade f é removido da *OpenList* (linha 8). Se um nó de melhor prioridade foi encontrado anteriormente (linha 9), o nó atual não pode ser parte do caminho ótimo e então é descartado. Caso contrário, o nó é movido para a lista *ClosedList* (linha 10). Durante a fase de expansão (linhas 11-15), para cada nó vizinho *neighbor* (linha 11), é verificado se algum outro nó com a mesma coordenada, mas com uma melhor prioridade já foi encontrado na *OpenList* ou *ClosedList* (linha 12). Caso o nó satisfaça essas condições, ele é adicionado à *OpenList* (linha 13). Caso contrário, o nó não pode pertencer ao um caminho ótimo e não é adicionado à nenhuma lista. Após esta etapa, o processo de busca é reiniciado até que a condição de parada seja verdadeira.

Quando o processo de busca é interrompido e a lista *OpenList* está vazia, significa que não existe um caminho entre as coordenadas iniciais C_i e C_f (linha 19). Caso contrário, o nó n_e de melhor prioridade f da *OpenList* possui as coordenadas C_f. Este nó possui um valor g que representa o valor do caminho ótimo entre as coordenadas C_i e C_f. É possível encontrar o caminho ótimo utilizando uma função de *backtrace* a partir do nó n_e.

Para garantir a optimalidade é necessário que algumas condições sejam satisfeitas. A primeira condição é que $h(N)$ seja um heurística admissível, ou seja, seja uma heurística que nunca superestime o custo de atingir o objetivo [109]. Sendo $g(N)$ o custo real para atingir o nó atual, temos, pela definição de $f(N)$ na Equação 2.17 que $f(N)$ nunca irá superestimar o verdadeiro custo de uma solução ao longo do caminho atual através de N.

Uma segunda condição necessária é a monotocidade, ou consistência. Esta condição é um pouco mais forte que a primeira e diz que $h(N)$ será consistente se, para cada nó N e todo sucessor N_s de N gerado por uma ação a, o custo estimado de alcançar o objetivo de N não é maior que o custo de se chegar a N_s somado ao custo estimado de N_s, como mostra a Equação 2.18:

$$h(N) \leq c(N, a, N_s) + h(N_s) \tag{2.18}$$

É possível demonstrar que toda heurística consistente também é admissível, e o algoritmo A-Star é ótimo se $h(N)$ for consistente [109].

2.3.2.2 A-Star Aplicado ao Alinhamento Múltiplo

Para que o A-Star seja aplicado ao problema do alinhamento múltiplo, é necessário descrever o problema como um grafo. Para isso, considere k sequências, de comprimento $n_1, ..., n_k$. Então podemos criar um grafo G, onde cada nó deste grafo possui cardinalidade k, a coordenada inicial C_i é $(0, ..., 0)$ e a coordenada final C_f é $(n_1, ..., n_k)$. Portanto, encontrar o alinhamento ótimo das sequências consiste em encontrar o melhor caminho entre as coordenadas C_i e C_f.

Em 1989, Spouge [118] apresentou o primeiro uso do algoritmo A-Star para o problema do alinhamento múltiplo ótimo de sequências e na ocasião foi relatado que o crescimento do número de sequências, e o comprimento delas, logo limitavam a sua aplicabilidade. A função heurística empregada para estimar o valor de um nó N foi a $h_{2,all}(N)$, onde utiliza-se o alinhamento em pares de todas as sequências, pois a soma dos alinhamentos ótimos em pares não pode ser pior do que a soma do alinhamento múltiplo ótimo. Este fato já tinha sido observado por Carrillo e Lipman (Seção 2.3.1.1), sendo utilizado para a criação do seu limite, conforme Equação 2.11. Para isso, antes de iniciar a etapa de busca do A-Star, os alinhamentos ótimos par-a-par de todas as sequências são calculados utilizando o algoritmo de Needleman-Wunsch (Seção 2.2.1). Por exemplo, para o alinhamento de três sequências S_1, S_2 e S_3, são calculados 3 alinhamentos ótimos em pares para os seguintes pares de sequências: (S_1, S_2), (S_1, S_3) e (S_2, S_3) e então toda a matriz de programação dinâmica é mantida na memória para cada par. Para calcular o valor $h(n)$ para um nó de coordenada (x, y, z), são adicionados os valores das células (x, y), (x, z) e (y, z) das matrizes de programação dinâmica das sequências (S_1, S_2), (S_1, S_3) e (S_2, S_3), respectivamente. Então o valor $g(N)$ é calculado com a função soma dos pares e $f(N)$ é a soma dos valores $h(N)$ e $g(N)$ (Equação 2.17).

Em 1994, Ikeda, *et al.* [54], alteraram o A-Star para obter um alinhamento heurístico, porém em tempo 314 vezes mais rápido. Outra contribuição deste trabalho é a criação de uma heurística ligeiramente mais forte do que a $h_{2,all}(N)$. Esta heurística utiliza um alinhamento não-ótimo para estimar e reduzir ainda mais o espaço de busca.

Em 1997, Shibuya *et al.* [113] propuseram uma estratégia para obter alinhamentos sub-ótimos, quando não for possível obter o ótimo, que possuem valores muito próximos ao ótimo. Este programa foi comparado com o MSA 2.0 e os autores concluíram que seus resultados eram muitos próximos.

Em 1998, Kobayashi, *et al.* [63] apresentaram estimadores mais fortes que os anteriores: as heurísticas $h_{2,all}(N)$ que utilizavam informações dos alinhamentos em pares. Neste trabalho, é estimado $h_{m,all}(N)$, onde k é o número de sequências, e m é algum número menor ou igual a k. O uso de um conjunto de 3 sequências é conhecido como heurística

$h_{3,one}$, e esta heurística requer uma quantidade muito maior de memória que a heurística anterior e até hoje, não é prático o uso dessa heurística para valores de $k > 3$.

Em 1999, Reinert, *et al.* [107] apresentaram o programa OMA (*Optimal Multiple Alignment*) que utiliza uma nova heurística. Os algoritmos anteriores exigiam que os alinhamentos em pares (ou trios) fossem realizados antes do início da busca do A-Star. Neste trabalho foi proposta uma heurística adaptativa onde os valores da heurística são calculados à medida que o espaço de busca é expandido, e não previamente, antes do início do algoritmo, diminuindo o número de cálculos necessários.

Lermen, *et al.* [77] apresentaram em 2000 o algoritmo GSA (*Goal directed Sequence Alignment*), uma nova versão do OMA, que foi comparado com outras 2 versões de programas para obter o alinhamento múltiplo ótimo. Neste algoritmo, foram adicionados *affine-gaps*, que aumentam consideravelmente o tempo de execução do programa e também fizeram um comparação direta do Carrillo-Lipman *bound* e A-Star, provando que o limite Carrillo-Lipman não pode descartar mais células do espaço de busca do que o algoritmo A-Star. O uso de memória é uma das principais limitações do A-Star. Durante a execução, após gerar um nó, ele é colocado em uma lista e, até o final da execução, um nó pode ser marcado como fechado e movido para uma lista de nós fechados, mas permanece na memória durante toda a execução. Para o caso do alinhamento múltiplo, o número de vizinhos de um nó é exponencial em relação ao número de sequências, e assim a memória pode ser o principal fator limitante.

Desta forma, nos anos seguintes encontramos algumas variações do A-Star para a solução do problema do alinhamento múltiplo. Normalmente, são combinadas mais de uma dentre as seguintes melhorias: (a) paralelização [37, 89]; (b) busca em profundidade e não a busca em largura [89, 112, 139]; (c) uso do Frontier A-Star [64, 89], variante do A-Star que não armazena os nós fechados, apenas os nós abertos de forma que seja encontrado apenas o escore do melhor caminho; e (d) *Delayed Duplicate Detection* (DDD) [37, 65, 66] que reduz o uso de memória, pois os novos nós gerados são armazenados em uma fila para serem processados em um passo futuro, já resolvendo conflitos de reexpansão de nós e também utiliza memória externa [37], comumente disponível em quantidade maior do que a memória RAM.

Uma das variantes do A-Star chamada de Partial Expansion A* (PEA*) [137] reduz a memória necessária gerando apenas nós sucessores mais promissores. Tipicamente, algoritmos de busca geram todos os nós sucessores de um nó, muitos deles possuem um valor f superior ao custo da solução ótima. Esses nós ocupam espaço na lista de nós abertos e nunca são expandidos. O algoritmo PEA* reduz a memória necessária para armazenar a lista de nós abertos, de forma que, dada uma constante C, todos os nós que possuem valor $f \leq C$ ao serem gerados não são inseridos na lista de nós abertos,

mas quem é inserido é o nó pai, com o valor f igual ao menor valor entre os nós que não foram inseridos. Durante uma fase de reexpansão, todos os nós são inseridos na lista de nós prioritários. Desta forma, caso $C = \infty$, o PEA* é equivalente ao A-Star e não reexpande nenhum nó. Caso $C = 0$, apenas nós com prioridade igual ao pai serão gerados e normalmente isso leva ao máximo *overhead* de reexpansão. Yoshizumi, *et al.* [137] mostraram que o problema do alinhamento múltiplo de sequências pode ter a memória bastante reduzida com o uso desta técnica.

Uma variante da técnica DDD é a *Hash-Based Delayed Duplicate Detection* (HBDDD) [67] que utiliza uma função *hash* para poder armazenar nós em arquivos. Como nós duplicados sempre possuem o mesmo valor de *hash*, eles serão armazenados no mesmo arquivo. Este algoritmo possui duas fases: uma de expansão e uma de *merge*. Durante a fase de expansão, todos os nós que possuem o mínimo atual conhecido da solução f_{min} são expandidos e armazenados em seus arquivos respectivos. Se um nó gerado possui $f \leq f_{min}$ então ele é expandido imediatamente, ao invés de ser armazenado em disco, procedimento conhecido como expansão recursiva. Quando todos os nós com f_{min} são expandidos a fase de *merge* é iniciada, de forma que cada arquivo é lido para uma *hash-table* e assim nós duplicados são removidos em tempo linear.

2.3.3 Comparação Múltipla Heurística

Como visto na Seção 2.3.1, o alinhamento global múltiplo ótimo demanda muitos recursos computacionais. Por isso, métodos heurísticos foram criados para obter alinhamentos múltiplos. O desafio desses métodos consiste em utilizar uma combinação apropriada de pesos de sequências, matrizes de escore e penalidade por *gaps* de forma que um bom alinhamento possa ser encontrado [87].

2.3.3.1 Métodos Progressivos

Algoritmos progressivos são baseados na ideia de construir algum alinhamento com um conjunto de sequências que possuem maior semelhança e então outras sequências são adicionadas ao alinhamento. O processo continua até que todas as sequências tenham sido consideradas [87].

A relação entre as sequências é geralmente estabelecida através de uma árvore filogenética, onde as sequências são comparadas em pares. Nessas árvores, as folhas são sequências que possuem maior similaridade.

Uma desvantagem dos métodos progressivos é a dependência do alinhamento em pares inicial. Assim, se o alinhamento inicial for um bom alinhamento, poucos erros acontecerão. Mas, se isso não acontece, muitos erros iniciais se propagarão para os próximos alinha-

mentos, o que pode resultar em um alinhamento global final com vários erros, ocasionando em um escore bem menor do que o obtido pelo algoritmo exato.

O Clustal [46] é um algoritmo bastante popular baseado em um método progressivo e utilizado desde 1988. Ele sofreu diversas mudanças ao longo dos anos buscando a melhoria do alinhamento múltiplo. ClustalW [129] é uma versão posterior ao Clustal, sendo que W significa *"weighting"*, representando a habilidade do programa de atribuir pesos para as sequências utilizadas.

O ClustalW é executado em 3 fases. Na fase 1, um alinhamento em pares de todas as sequências é realizado. Em seguida, na fase 2, os escores dos alinhamentos em pares são utilizados para a criação de uma árvore filogenética. Por último, na fase 3, o alinhamento múltiplo das sequências é realizado utilizando um algoritmo de programação dinâmica guiado pela árvore produzida na fase 2. Dessa forma, as sequências mais semelhantes são alinhadas e em seguida as outras, ou grupos de sequências, são adicionadas e, guiados pelo alinhamento inicial, é produzido um alinhamento múltiplo global.

T-Coffee [90] é um outro programa de alinhamento progressivo que usa um sistema de pesos nas posições das sequências para gerar um alinhamento múltiplo que é mais consistente que o alinhamento em par de todas as sequências [87]. No T-Coffee, os dados são organizados em uma biblioteca primária e uma biblioteca estendida. A biblioteca primária armazena informações sobre todos os alinhamentos par-a-par das sequências de entrada. Logo existem $(n \times (n-1))/2$ elementos nessa biblioteca. Cada elemento da biblioteca primária possui várias entradas com informações sobre o alinhamento global par-a-par e os 10 melhores alinhamentos locais sem sobreposição. A biblioteca primária é estendida da seguinte maneira. Cada entrada é comparada com todas as outras entradas e o resultado dessa comparação é um peso, que descreve o grau no qual essa entrada é consistente com as outras entradas. Tanto a biblioteca primária como a biblioteca estendida podem ser utilizadas para a obtenção do alinhamento múltiplo.

2.3.3.2 Métodos Iterativos

Os métodos iterativos buscam resolver o principal problema existente nos métodos progressivos, onde erros nos alinhamentos iniciais possuem um grande peso e são propagados para o alinhamento múltiplo final. Buscando melhorar o alinhamento em modo geral, ou seja, melhorando o escore do alinhamento, os métodos iterativos buscam corrigir esse problema realinhando repetidamente os subgrupos de sequências e então alinhando esses subgrupos em um alinhamento global de todas as sequências.

O DIALIGN é um método iterativo que realiza o alinhamento múltiplo de sequências sem adicionar penalidade para *gaps* [85]. No DIALIGN, o alinhamento é um conjunto ordenado de diagonais onde as diagonais são alinhamentos de subsequências sem *gaps*

(somente *matches* e *mismatches*). Este algoritmo é executado em três fases. Na primeira fase, todos os pares de alinhamentos DIALIGN são calculados, isto é, $n(n\text{-}1)/2$ cálculos, um para cada alinhamento, onde n é o número de sequências. Na segunda fase, as diagonais que compõem o alinhamento em pares são ordenadas pelo escore e grau de sobreposição com outras diagonais. Essa lista ordenada é usada para obter um alinhamento múltiplo com um algoritmo guloso. Na última fase, este alinhamento é completado com um procedimento iterativo onde partes das sequências que ainda não foram alinhadas com A são realinhadas executando a fase 2 novamente, de forma que diagonais consistentes não alinhadas sejam incluídas em A. Esta fase é repetida até que nenhuma diagonal com peso positivo possa ser incluída.

Para melhorar a qualidade dos alinhamentos produzidos, o DIALIGN-TX [121] foi proposto. Como em todas as versões anteriores, a saída da primeira fase é um conjunto de diagonais de escore alto. Essas diagonais são usadas no DIALIGN-TX para construir uma árvore guia na fase 2. Na fase 3, dois métodos são usados para gerar 2 alinhamentos: um pelo método progressivo e o outro pelo método original do DIALIGN. Ambos alinhamentos são avaliados e o melhor é mantido.

2.3.3.3 Outros Métodos Heurísticos

A maioria dos métodos para alinhamentos múltiplos normalmente determina a semelhança entre todos os pares de sequências que devem ser comparados. Outros métodos, como a aproximação por grupos [87], utilizam um consenso entre cada grupo de sequências e esse consenso é utilizado para futuro alinhamento entre grupos. Exemplos de programas com essa abordagem são o PIMA [115] e o MULTAL [128]. Alguns desses métodos utilizam a distância de uma árvore filogenética para organizar as sequências e as duas mais próximas são alinhadas. O alinhamento consenso obtido é alinhado com outra sequência, conjunto ou outro consenso até se obter um alinhamento com todas sequências [87].

Capítulo 3

Alinhamento Secundário de RNA

A previsão da estrutura secundária de uma sequência de RNA é uma das operações mais importantes para estudar as características biológicas deste tipo de sequência [27]. Esta operação possui como resultado um escore que, no caso mais simples, representa o número de pares de bases existentes na sequência e também uma estrutura secundária ilustrando quais nucleotídeos estão pareados.

Quando se deseja comparar duas sequências de RNA, frequentemente considera-se não apenas as similaridades entre a estrutura primária das sequências, mas também a estrutura secundária de cada uma delas. Para isso, pode-se obter um alinhamento secundário ou alinhamento estrutural de sequências de RNA. O resultado dessa operação é um alinhamento de sequências e uma estrutura secundária comum a elas. Neste capítulo, será detalhado o problema da previsão da estrutura secundária de uma ou mais sequências de RNA.

3.1 Estrutura Secundária

Um RNA (*ribonucleic acid*) é um polímero de nucleotídeos [87] representados pelo alfabeto $\Sigma = \{A, U, G, C\}$ (Seção 2.1). Diferente do DNA e das proteínas, os nucleotídeos de uma sequência de RNA podem formar pares de bases. Estas ligações podem ser entre os pares G-C/C-G e A-U/U-A, chamadas de base de Watson-Crick [70]. Porém outros pares de bases também podem ser formados, chamados de Emparelhamento Wobble [19], sendo o mais comum o emparelhamento de G-U/U-G [70]. Os ncRNAs estão envolvidos em uma variedade de processos: regulação dos genes codificantes (microRNAs [22]), biosíntese de proteínas (transfer RNAs) [13, 138], remoção de "introns" (small nuclear RNA) [28],entre outros.

A estrutura secundária do RNA é definida como o conjunto de pares de bases que podem ser mapeados em um plano [27]. A Figura 3.1 ilustra duas notações para a estrutura

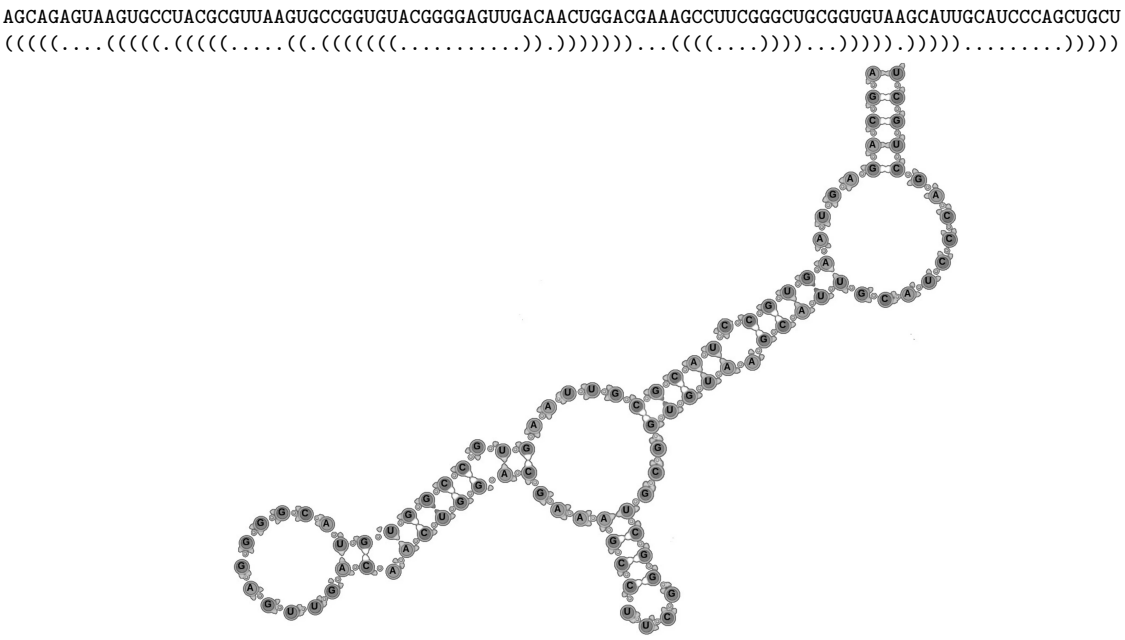


Figura 3.1: Estrutura secundária de um RNA. A sequência e a notação *dot bracket* da estrutura secundária é mostrada no topo. Abaixo, um diagrama da estrutura secundária é apresentado.

secundária. A primeira, ao topo, é a notação *dot bracket* onde parênteses balanceados indicam nucleotídeos pareados e um ponto indica um nucleotídeo não pareado. A segunda forma é o diagrama da estrutura secundária, uma representação gráfica da figura em um plano que mostra claramente os nucleotídeos pareados.

O RNA possui uma grande variedade de tamanhos, que podem variar de poucas bases (aproximadamente 20, como microRNAs) até dezenas de milhares de bases, como X-inactive specific transcript (XIST) RNA. Embora a estrutura secundária seja uma representação simplificada de uma estrutura tridimensional bem mais complexa, ela ainda captura os elementos mais importantes da estrutura completa [80] e, por essa razão ela é a base para o estudo do RNA [27].

Ao se considerar uma estrutura secundária, geralmente não é importante saber qual nucleotídeo específico é encontrado em uma determinada posição, mas é muito importante saber que o nucleotídeo de uma posição pode formar um par de base com um nucleotídeo de uma outra posição, criando então uma determinada estrutura. Mudanças nas estruturas primárias que não alteram os padrões dos pares de bases estão raramente envolvidos no processo de evolução [27], mas mudanças nas sequências primárias que alteram os pares de bases irão afetar o funcionamento da região em questão.

Isto leva a um padrão de conservação dos nucleotídeos no RNA, de forma que este é muito mais conservado na estrutura do que na sequência [27]. Mudanças na estrutura primária podem ser compensatórias, isto é, quando ocorre a mudança de dois pares de

bases pareados por outros dois nucleotídeos que também formam um pareamento. Sendo assim, o pareamento não é alterado e, consequentemente, a estrutura também não.

Por exemplo, considere que uma sequência possui o pareamento A-U em um determinado ncRNA. Durante o processo de evolução pode ocorrer a substituição desse par por um G-C, que também forma um par de bases. Assim, apesar de haver diferenças entre as sequências, a função desta região permanece inalterada. Tais alterações sugerem que as ferramentas que consideram apenas a estrutura primária podem ser inadequadas para a análise de regiões onde alterações compensatórias ocorrem.

3.1.1 Principais Estruturas

O pareamento de bases cria regiões que possuem interações comuns entre diferentes sequências de RNA e para essas regiões existe uma convenção de nomes. As regiões mais comuns estão representadas na Figura 3.2.

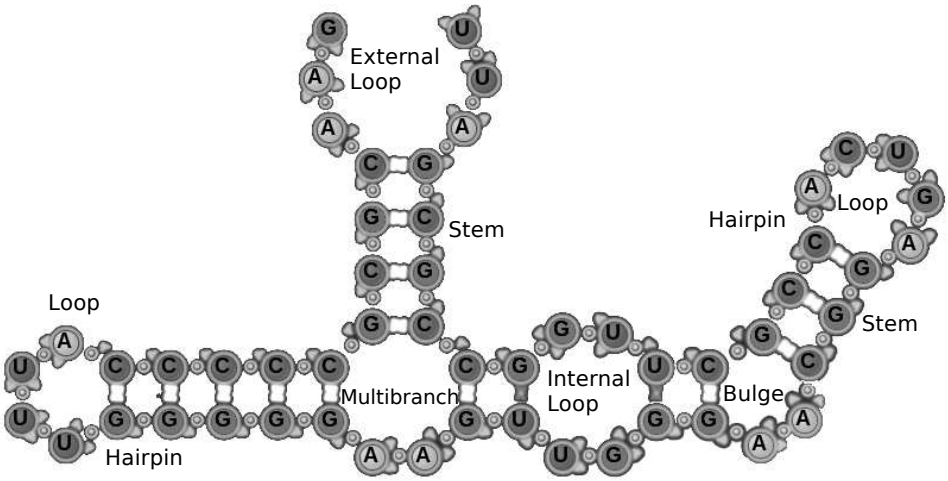


Figura 3.2: Tipos de regiões (nomes em inglês) que compõem as regiões de uma estrutura secundária de um RNA.

- **Stem (Talo):** Uma região que possui um ou mais nucleotídeos pareados consecutivamente;
- **Loop (Laço):** Quando dois nucleotídeos estão pareados e entre eles existem consecutivos nucleotídeos não pareados;
- **Hairpin (Grampo):** Combinação de uma região que possui um *stem* e uma que possui um *loop*. Na Figura 3.2 são apresentadas dois *hairpins*, um à esquerda e um à direita;
- **External Loop (Laço Externo):** Região não-pareada que contém o final da sequência de RNA;

- **Internal Loop (Laço Interno):** Região com dois *stems* adjacentes, ou um único par de bases, contendo nucleotídeos não pareados entre eles;
- **Bulge (Barriga):** Caso específico do Loop Interno, quando os nucleotídeos não pareados que compõem a região estão em apenas um dos lados da sequência de RNA;
- **Multibranch (Junção):** Também chamada de *multibranch-loop*. Região que contém nucleotídeos não-pareados e que combina outras sub-regiões em estruturas mais complexas.

As regiões discutidas acima compõem a estrutura secundária básica, porém existem outros tipos de interações, como os *pseudo-knots*. Por exemplo, na Figura 3.2 são conhecidas interações entre as regiões do *loop* do *hairpin* à esquerda, com o *loop* interno representado à direita [51]. Algumas vezes essas interações não podem ser representadas na estrutura secundária, mas mostram que apesar desta estrutura representar visualmente um RNA e muitas informações sobre o seu formato, algumas vezes não é capaz de representar todas as interações existentes em um RNA, sendo necessária a análise da estrutura 3D.

3.2 Predição de Estrutura Secundária de RNA

Uma das tarefas mais básicas ao trabalhar com RNA é tentar prever a estrutura secundária de uma determinada sequência. Neste caso, a entrada do algoritmo é uma sequência de RNA, e a saída é a estrutura secundária, seja através da notação de parênteses, ou uma representação gráfica da estrutura secundária como ilustrado na Figura 3.2.

Para prever uma estrutura secundária, o algoritmo mais simples pode verificar todas as possibilidades de combinações entre os nucleotídeos da sequência, verificando quais formam pares de bases e buscando maximizar o número de pares de bases. Porém desta forma é necessário realizar muitas combinações e o número de possibilidades crescem exponencialmente em relação ao comprimento da sequência.

Para simplificar esta situação, pode-se utilizar programação dinâmica [16]. Desta forma, nos primeiros passos do algoritmo pares de bases são criados, e nos passos seguintes novos pares de bases são adicionados, ou estruturas que possuam muitos pares de bases são combinadas. Nesta seção, descrevemos o algoritmo de Nussinov, que realiza a predição da estrutura secundária de uma única sequência.

3.2.1 Algoritmo de Nussinov

Nussinov propôs o primeiro algoritmo [91] para obter a estrutura secundária de uma sequência de RNA. O objetivo desse algoritmo é encontrar o maior número de pares de bases em uma sequência de RNA e isto é feito utilizando-se uma matriz de programação dinâmica. Apesar do algoritmo de Nussinov ser muito simples para predizer com acurácia estruturas de RNA, seu princípio básico é amplamente utilizado em algoritmos mais complexos. O algoritmo de Nussinov procura a melhor estrutura existente entre a posição i e j, $(i < j)$ e as iterações consistem em aumentar o i e diminuir o j. Para isso em cada posição deve-se considerar quatro possibilidades, como apresentado na Figura 3.3.

Formalmente, define-se a equação de recorrência do algoritmo, considerando $D(i, j)$ como o maior número de pares de bases para a subsequência $S_{i,j}$ [91], $D_1(i)$ o i-ésimo nucleotídeo da primeira sequência e $s(a, b)$ a função de escore que retorna 1 se os nucleotídeos a e b formam um par-de-bases e 0 caso contrário.

$$D(i,j) = \max \begin{cases} D(i, j-1) \\ D(i+1, j) \\ D(i+1, j-1) + s(S_1(i), S_2(j)) \\ \max_{i<k<j-1}(D(i,k) + D(k+1, j)) \end{cases} \tag{3.1}$$

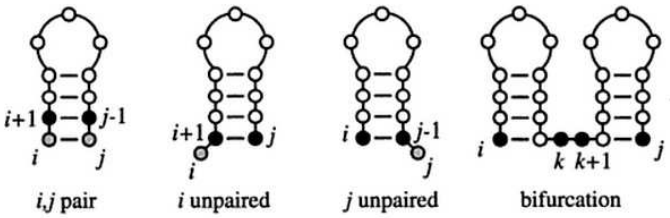


Figura 3.3: Quatro possibilidades da equação de recorrência do algoritmo de Nussinov [27].

A Equação 3.1 representa possíveis situações:

- Adicionar o par i, j na melhor estrutura para a subsequência $i + 1, j - 1$.
- Adicionar uma base não-pareada na posição i para a melhor estrutura encontrada para a subsequência $i + 1, j$.
- Adicionar uma base não-pareada na posição j para a melhor estrutura encontrada para a subsequência $i, j - 1$.
- Combinar duas subestruturas ótimas i, k e $k + 1, j$.

Além disso, a Equação 3.2 representa a inicialização das duas primeiras diagonais da matriz de programação dinâmica, considerando uma sequência de tamanho n:

$$\begin{aligned} D(i,i) &= 0,\ 1 \leq i \leq n \\ D(i,i-1) &= 0,\ 2 \leq i \leq n \end{aligned} \tag{3.2}$$

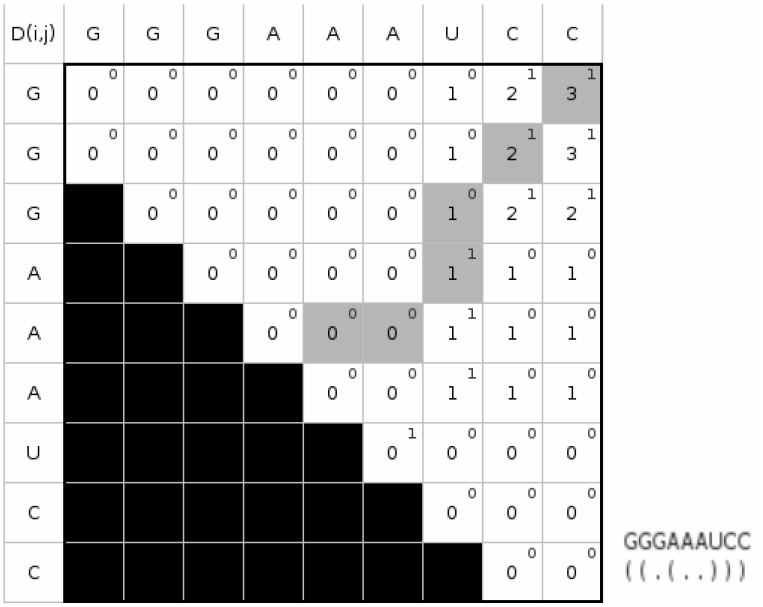


Figura 3.4: Matriz de programação dinâmica do Algoritmo Nussinov para a Sequência $S_1 = GGGAAAUCC$. As células em cinza foram as células percorridas durante a etapa de *traceback*.

A matriz de programação dinâmica para a sequência $GGGAAAUCC$ está ilustrada na Figura 3.4. Nela, cada célula possui o melhor valor de pares de bases para uma determinada subsequência e desta forma a resposta é obtida através de soluções de instâncias menores do problema. Como os algoritmos de comparação primária de sequências, este algoritmo também possui uma segunda fase de *traceback*, onde, a partir da célula na posição final, é verificado qual célula foi utilizada para produzir o melhor valor. Desta forma, sabe-se se o nucleotídeo da posição está pareado, não está pareado, ou é o ponto de junção de duas subestruturas ótimas.

3.2.2 Algoritmo de McCaskill

O algoritmo de Nussinov foi o primeiro a apresentar uma solução para o cálculo da estrutura secundária de um RNA. Porém, em sua versão mais simples, ele apenas tenta maximizar os pares de bases da estrutura, o que muitas vezes não representa uma estrutura biologicamente relevante. McCaskill propôs um algoritmo [81] combinando uma

análise probabilística que também leva em conta a energia da estrutura secundária, visando melhorar a modelagem do modelo biológico. O algoritmo baseia-se na função de partição Z representada na equação 3.3.

$$Z = \sum_{P} \exp(-e(P)/RT) \tag{3.3}$$

Nessa equação, $e(P)$ é a energia de uma possível estrutura P, R é a constante de gás e T é a temperatura. O algoritmo de McCaskill calcula a função de partição para todas as subestruturas que podem ser formadas pela sequência S. Considerando que cada par de bases contribui com uma quantidade de energia e_{bp} para o alinhamento, independente do seu contexto, são geradas duas matrizes de programação dinâmica Q e Q^{bp}, dadas pelas equações 3.4 e 3.5.

$$Q_{i,j} = Q_{i,j-1} + \sum_{i \leq k < j-l} (Q_{i,k-1} Q^{bp}_{k,j}) \tag{3.4}$$

$$Q^{bp}_{i,j} = \begin{cases} Q_{i+1,j-1} \exp(\frac{-e_{bp}}{RT}) & \text{se } S_i, S_j \text{ podem formar um par de bases} \\ 0 & \text{caso contrário.} \end{cases} \tag{3.5}$$

A célula $Q_{i,j}$ da matriz de programação dinâmica contém a função de partição para a subsequência $S[i..j]$, já $Q^{bp}_{i,j}$ contém a função de partição dado que os nucleotídeos da posição i e j podem formar um par de bases. A função de partição geral Z encontra-se na célula final $Q_{1,n}$.

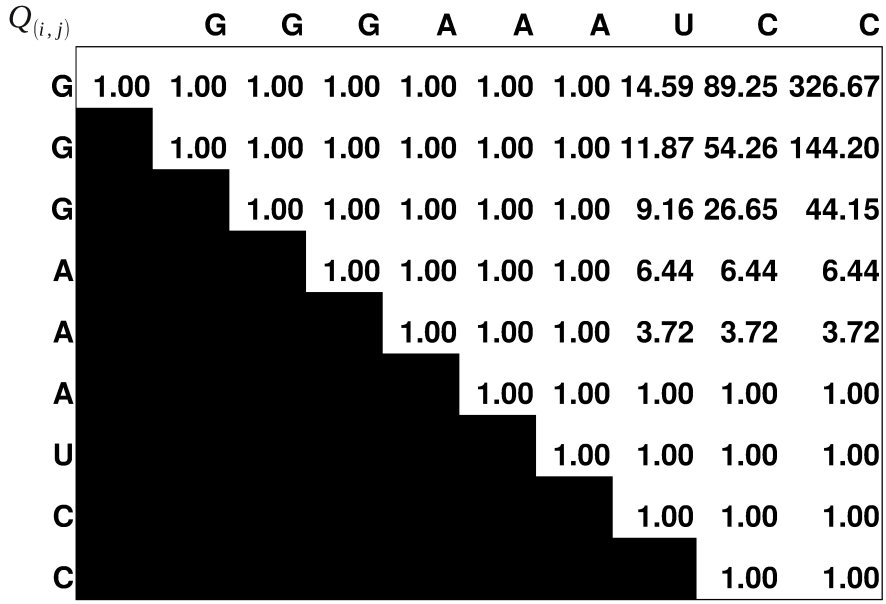
As Figuras 3.5 e 3.6 ilustram a matriz de programação dinâmica Q e Q^{bp} para a sequência $S = GGGAAAUCC$. Dada essas duas matrizes Q e Q^{bp} é possível calcular a matriz de probabilidade de pares de bases individuais (i, j), conforme a equação 3.6.

$$P^{bp}_{i,j} = \sum_{P \ni (i,j)} (\exp(\frac{-e(P)}{RT}))/Z \tag{3.6}$$

Para calcular esse valor para todos os possíveis pares de bases de uma sequência, utiliza-se a equação de recorrência 3.7 para calcular a matriz P^{bp}.

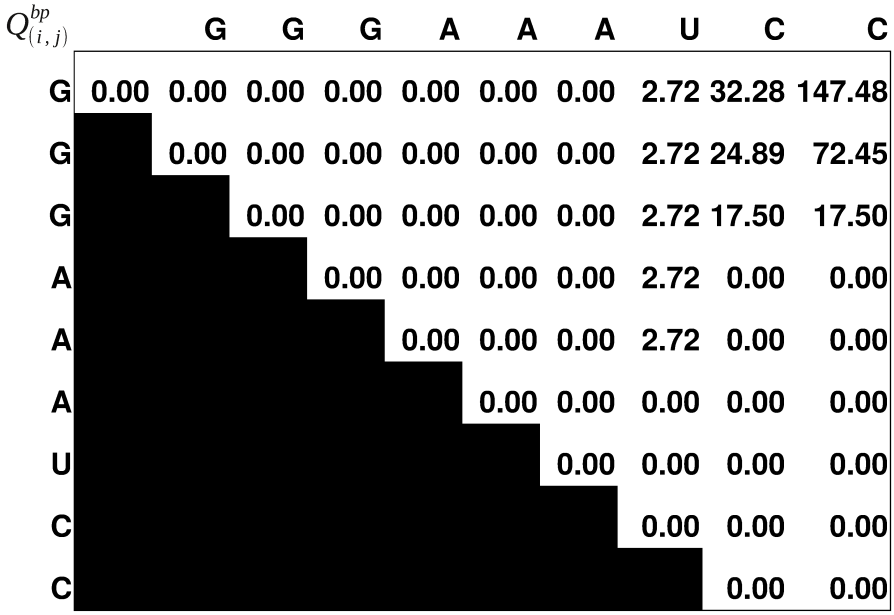
$$P^{bp}_{i,j} = \frac{Q_{1,i-1} Q^{bp}_{i,j} Q_{j+1,n}}{Q_{1,n}} + \sum_{p<i,j<q} P^{bp}_{p,q} \frac{\exp(\frac{-e_{bp}}{RT}) Q_{p+1,i-1} Q^{bp}_{i,j} Q_{j+1,q-1}}{Q^{bp}_{p,q}} \tag{3.7}$$

A Figura 3.7 ilustra a matriz de programação dinâmica P para a sequência $S = GGGAAAUCC$. Nela, a célula (i, j) indica a probabilidade do nucleotídeo i e j estarem pareados.



$Q_{(i,j)}$	G	G	G	A	A	A	U	C	C
G	1.00	1.00	1.00	1.00	1.00	1.00	1.00	14.59	89.25	326.67
G		1.00	1.00	1.00	1.00	1.00	1.00	11.87	54.26	144.20
G			1.00	1.00	1.00	1.00	1.00	9.16	26.65	44.15
A				1.00	1.00	1.00	1.00	6.44	6.44	6.44
A					1.00	1.00	1.00	3.72	3.72	3.72
A						1.00	1.00	1.00	1.00	1.00
U							1.00	1.00	1.00	1.00
C								1.00	1.00	1.00
C									1.00	1.00

Figura 3.5: Matriz de programação dinâmica Q do algoritmo de McCaskill.

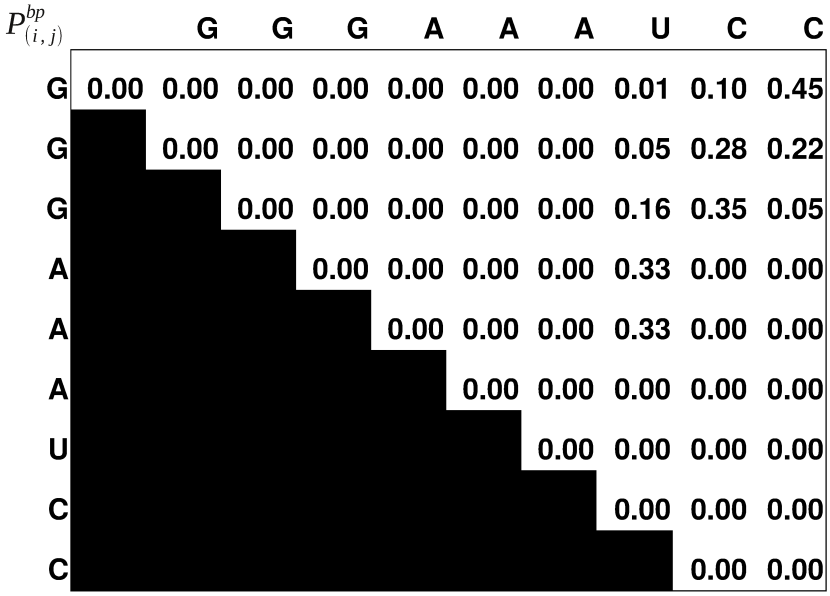


$Q^{bp}_{(i,j)}$	G	G	G	A	A	A	U	C	C
G	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.72	32.28	147.48
G		0.00	0.00	0.00	0.00	0.00	0.00	2.72	24.89	72.45
G			0.00	0.00	0.00	0.00	0.00	2.72	17.50	17.50
A				0.00	0.00	0.00	0.00	2.72	0.00	0.00
A					0.00	0.00	0.00	2.72	0.00	0.00
A						0.00	0.00	0.00	0.00	0.00
U							0.00	0.00	0.00	0.00
C								0.00	0.00	0.00
C									0.00	0.00

Figura 3.6: Matriz de programação dinâmica Q^{bp} do algoritmo de McCaskill.

3.3 Alinhamento e Dobramento Simultâneo: Comparação do RNA para mais de uma Sequência

Quando são consideradas duas ou mais sequências de RNA e deseja-se obter uma estrutura que seja comum a elas, existem diferentes abordagens que podem ser utilizadas para a obtenção do alinhamento de sequências e uma estrutura secundária comum



$P^{bp}_{(i,j)}$		G	G	G	A	A	A	U	C	C
G	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.10	0.45
G		0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.28	0.22
G			0.00	0.00	0.00	0.00	0.00	0.16	0.35	0.05
A				0.00	0.00	0.00	0.00	0.33	0.00	0.00
A					0.00	0.00	0.00	0.33	0.00	0.00
A						0.00	0.00	0.00	0.00	0.00
U							0.00	0.00	0.00	0.00
C								0.00	0.00	0.00
C									0.00	0.00

Figura 3.7: Matriz de programação dinâmica P^{bp} do algoritmo de McCaskill.

(alinhamento estrutural).

A Figura 3.8 ilustra a classificação de Gardner e Giegerich [25] destas diferentes abordagens. A primeira estratégia "A", utiliza um alinhamento primário de sequências para depois criar uma estrutura secundária comum a elas. Este método funciona bem quando existe uma alta similaridade na estrutura primária.

A estratégia "B" engloba os métodos estilo Sankoff, que são o objetivo desta Tese e resolvem o problema do alinhamento e dobramento simultâneo. Dentre as diferentes abordagens mostradas, a estratégia "B" possui a maior complexidade computacional e frequentemente é utilizada apenas para o alinhamento em pares.

A estratégia "C" gera uma estrutura secundária para cada sequência. As estruturas secundárias são combinadas para gerar a estrutura consenso. Este método depende de uma alta confiança na estrutura secundária gerada para cada sequência.

A seguir, discorremos sobre a estratégia "B". Algoritmos para a solução do problema de alinhamento e dobramento simultâneo recebem como entrada $n > 2$ sequências de RNA e produzem como saída uma estrutura secundária e um alinhamento. A Figura 3.9 mostra um exemplo de uma solução de um alinhamento estrutural de duas sequências, sendo a resposta composta por um alinhamento de sequências, com *matches*, *mismatches* e *gaps*, e também por uma notação de parênteses representando uma estrutura secundária comum às duas sequências.

A saída do algoritmo é produzida combinando um sistema de escores, onde são considerados escores dos alinhamentos das sequências e das estruturas. O algoritmo visa atribuir penalidade às substituições, inserções e deleções, porém reduz essa penalidade

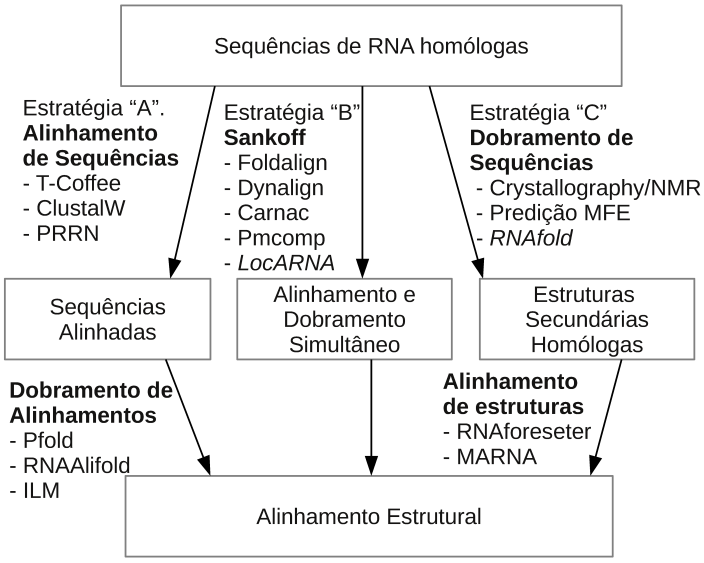


Figura 3.8: Classificação adaptada de Gardner e Giegerich [25] para as diferentes abordagens que podem ser utilizadas para a obtenção de um alinhamento de sequências e uma estrutura secundária comum. Em itálico, os exemplos de programas que nós adicionamos à classificação.

para substituições compensatórias, ou seja, substituições de nucleotídeos pareados mas que não alteram a estrutura.

Os conceitos de alinhamento global e alinhamento local, explicados na Seção 2.1.1, também são válidos. O alinhamento global utiliza todos os nucleotídeos nas sequências, enquanto no local, apenas subsequências são incluídas no alinhamento final.

O problema do alinhamento e dobramento simultâneo é resolvido por variantes do algoritmo de Sankoff. Este algoritmo utiliza uma matriz de programação dinâmica, com uma base no algoritmo de Nussinov (Seção 3.2.1), porém estendido para duas sequências.

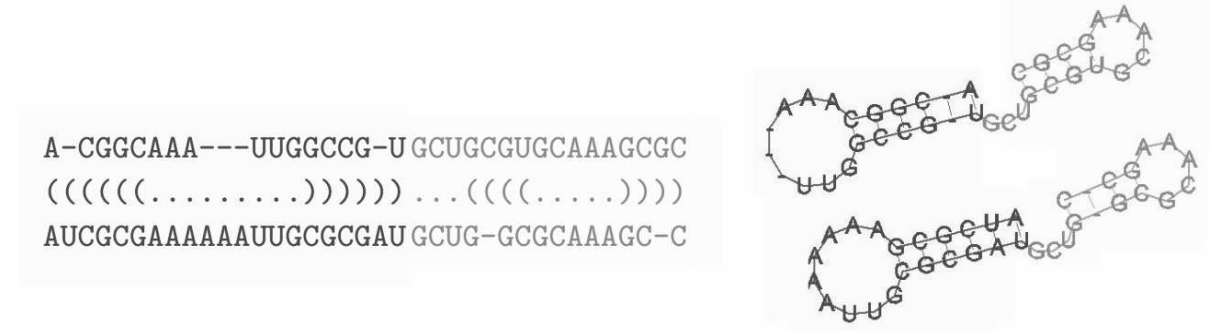


Figura 3.9: Um exemplo de um alinhamento e uma estrutura comum. Parênteses indicam bases pareadas. Pontos indicam bases não-pareadas. Observe que alguns *gaps* são adicionados nas sequências para chegar a um estrutura comum, representada à esquerda [141].

Podemos citar entre estes métodos o Foldalign [38], PMcomp [49], LocARNA [134]. Uma desvantagem desses métodos é a alta complexidade de tempo, $O(n^6)$, e memória, $O(n^4)$, onde o comprimento das sequências, considerando que as sequências possuem o mesmo tamanho n. Assim existe um consumo de recursos muito alto até mesmo para pequenas instâncias do problema.

Nesta seção iremos descrever o algoritmo de Sankoff e três métodos heurísticos baseados nele (Foldalign, PMcomp e LocARNA), todos utilizados para a obtenção do alinhamento estrutural em pares. Também será mostrado como os algoritmos podem ser combinados com outras estratégias para a obtenção de um alinhamento múltiplo estrutural.

3.3.1 Comparação Estrutural Múltipla

Da mesma forma que o alinhamento primário, o alinhamento secundário pode ser feito com pares de sequências (alinhamento em pares), ou com um conjunto de sequências (alinhamento múltiplo), onde nucleotídeos similares de um conjunto de n sequências ($n > 2$) são alinhados juntos. Gardner *et al.* [38] mostraram que, para o estudo de ncRNAs com similaridade inferior a 60%, é necessário o uso de informações da estrutura secundária enquanto se faz o alinhamento.

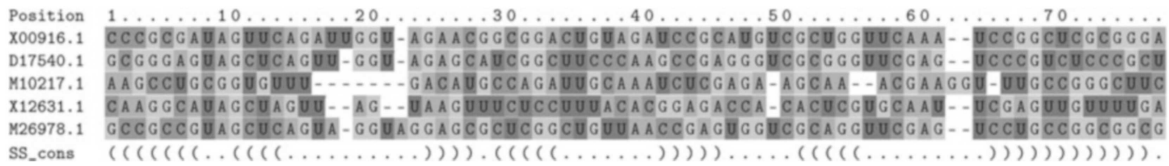


Figura 3.10: Alinhamento múltiplo de sequências de RNA [27].

A Figura 3.10 ilustra um alinhamento múltiplo de sequências de RNA. Na parte de cima da figura, é possível ver o alinhamento com os *matches*, *mismatches* e *gaps*. Na última linha é possível ver a estrutura secundária comum a elas. Ela também é chamada de estrutura de consenso, e diz a estrutura comum mais próxima entre as múltiplas sequências que compõem o alinhamento.

3.3.2 Algoritmo Exato - Sankoff

O algoritmo de Sankoff [111] visa fazer o alinhamento e dobramento simultâneo de múltiplas sequências de RNA, obtendo o resultado ótimo. Este algoritmo baseia-se na recursão do alinhamento de sequências (Equação 2.3) e a recursão para a predição da estrutura secundária (Equação 3.1).

Como visto na Seção 3.2.1, o algoritmo de Nussinov para fazer o dobramento de uma única sequência via uma equação de recorrência onde i é reduzido e j é aumentado ($i < j$). Para estender esse algoritmo para duas sequências, são mantidas as variáveis i e j que correspondem à primeira sequência. Porém, são adicionadas as variáveis k e l ($k < l$) para a segunda sequência.

A equação de recorrência do algoritmo de Sankoff é apresentada na equação 3.8. Todas as implementações atuais do algoritmo de Sankoff empregam variantes da mesma equação de recorrência [141]. Neste trabalho, utilizamos as equações empregadas no Foldalign [38] e FoldalignM [131].

$$
D_{i,j,k,l} = \max \begin{cases}
D_{i+1,j,k,l} + \gamma & \text{(a)} \\
D_{i,j-1,k,l} + \gamma & \text{(b)} \\
D_{i,j,k+1,l} + \gamma & \text{(c)} \\
D_{i,j,k,l-1} + \gamma & \text{(d)} \\
D_{i+1,j,k+1,l} + \sigma(S_1(i), S_2(k)) & \text{(e)} \\
D_{i,j-1,k,l-1} + \sigma(S_1(j), S_2(l)) & \text{(f)} \\
D_{i+1,j-1,k,l} + \beta_{ij}^{S_1} + 2.\gamma & \text{(g)} \\
D_{i,j,k+1,l-1} + \beta_{kl}^{S_2} + 2.\gamma & \text{(h)} \\
D_{i+1,j-1,k+1,l-1} + \beta_{ij}^{S_1} + \beta_{kl}^{S_2} + \tau(S_1(i), S_1(j), S_2(k), S_2(l)) & \text{(i)} \\
\max\limits_{\substack{i<m<j \\ k<n<l}} \{D_{i,m,k,n} + D_{m+1,j,n+1,l}\} & \text{(j)}
\end{cases} \tag{3.8}
$$

Na equação 3.8, $S_1(i)$ representa o i-ésimo caractere da primeira sequência, a função $\sigma(S_1(i), S_2(k))$ descreve a substituição de nucleotídeos não pareados e a função $\tau(S_1(i), S_1(j), S_2(k), S_2(l))$ descreve o custo de substituição compensatória do par de bases $S_1(i)$-$S_1(j)$ por $S_2(k)$-$S_2(l)$. $\beta_{ij}^{S_1}$ representa o custo de adicionar um par de bases na posição (i, j). Nesta equação, os termos (a) até (d) adicionam um nucleotídeo não pareado com um *gap*. O termo (e) e (f) adicionam um nucleotídeo não pareado. O termo (g) e (h) adicionam um par de base em uma sequência e um *gap* na outra sequência. O termo (i) adiciona um par de base em ambas as sequências e o último termo (j) junta duas subestruturas e uma para criar uma estrutura de *multibranch* (Seção 3.1.1).

O algoritmo de Sankoff para duas sequências é apresentado no Algoritmo 3. Ele recebe como parâmetro duas sequências, S_1 e S_2 de tamanhos n_1 e n_2, respectivamente. Considere também que $S_1(i)$ representa o caractere na i-ésima posição da primeira sequência. Além disso, a matriz de programação dinâmica D está pré-alocada em memória.

No Algoritmo 3 existem 6 *loops* "for" aninhados (linhas 1 a 4, 14 e 15), proporcionais

aos tamanhos das sequências. Os escores para *matches*, *mismatches*, pares de bases e substituições compensatórias são calculados nas linhas 5 a 13, usando a equação de recorrência (Equação 3.8), termos (a) a (i). Após isso (linhas 14 a 18), as regras de *multibranch* são aplicadas (Equação 3.8, termo (j)). Ao final, o maior valor entre todos os termos é salvo na posição (i, j, k, l) (linha 19).

Algoritmo 3 : `Algoritmo de Sankoff para 2 sequências`

1: **for** $i = n_1 \rightarrow 1$ **do**
2: **for** $k = n_2 \rightarrow 1$ **do**
3: **for** $j = i \rightarrow n_1$ **do**
4: **for** $l = k \rightarrow n_2$ **do**
5: $d \leftarrow D_{i+1,j,k,l} + \gamma$
6: $d \leftarrow \max(d, D_{i,j-1,k,l} + \gamma)$
7: $d \leftarrow \max(d, D_{i,j,k+1,l} + \gamma)$
8: $d \leftarrow \max(d, D_{i,j,k,l-1} + \gamma)$
9: $d \leftarrow \max(d, D_{i+1,j,k+1,l} + \sigma(S_1(i), S_2(k)))$
10: $d \leftarrow \max(d, D_{i,j-1,k,l-1} + \sigma(S_1(j), S_2(l)))$
11: $d \leftarrow \max(d, D_{i+1,j-1,k,l} + \beta_{ij}^{S_1} + 2.\gamma)$
12: $d \leftarrow \max(d, D_{i,j,k+1,l-1} + \beta_{kl}^{S_2} + 2.\gamma)$
13: $d \leftarrow \max(d, D_{i+1,j-1,k+1,l-1} + \beta_{ij}^{S_1} + \beta_{kl}^{S_2} + \tau(S_1(i), S_1(j), S_2(k), S_2(l)))$
14: **for** $m = i + 1 \rightarrow j$ **do**
15: **for** $n = k + 1 \rightarrow l$ **do**
16: $d \leftarrow \max(d, D_{i,m,k,n} + D_{m+1,j,n+1,l})$
17: **end for**
18: **end for**
19: $D_{i,j,k,l} \leftarrow d$
20: **end for**
21: **end for**
22: **end for**
23: **end for**

Admitindo que n_1 e n_2 são tamanhos próximos, vamos considerar n o tamanho das duas sequências para a análise da complexidade. Com 6 *loops* for aninhados, o algoritmo possui complexidade de tempo $O(n^6)$ e a complexidade da memória é da ordem de $O(n^4)$ [111]. É possível representar a matriz de programação dinâmica de Sankoff como uma matrix de 4 dimensões. Graficamente, podemos representar como várias matrizes, uma externa (ME) e um conjunto de matrizes internas (MI). Neste caso, cada célula da matriz externa é uma outra matriz de duas dimensões.

A figura 3.11 mostra as matrizes de programação dinâmica de Sankoff para duas sequências de 6 nucleotídeos, onde a matriz da direita é a matriz externa, com tamanho 5x5. Cada matriz interna possui um tamanho diferente, que é ilustrado pelo número da

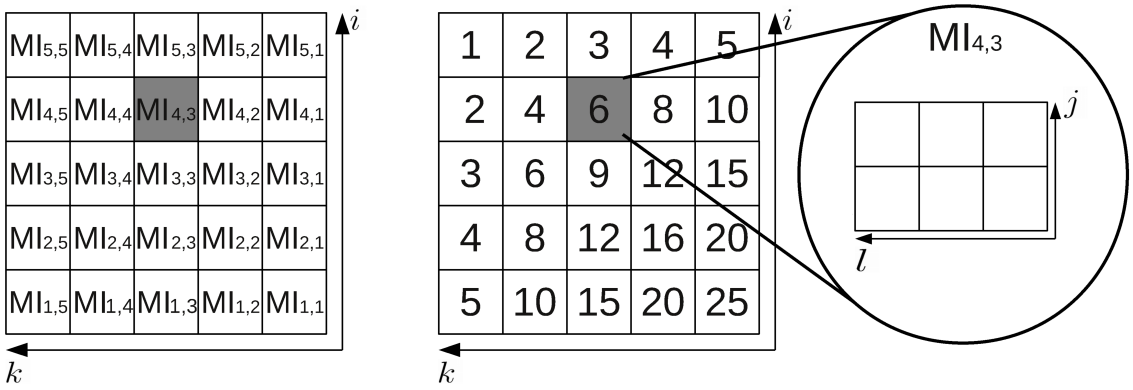


Figura 3.11: Representação gráfica da matriz de programação dinâmica de 4 dimensões do algoritmo de Sankoff para uma sequência de tamanho 5. A matriz da esquerda mostra as coordenadas de cada matriz interna. Do lado direito, o tamanho de cada matriz interna é mostrado. No detalhe, é possível ver a matriz interna $MI_{4,3}$ com 6 células.

matriz da direita. No detalhe, é possível ver a matriz $MI_{4,3}$ com 6 células ($S_{4,5,3,5}$, $S_{4,5,3,4}$, $S_{4,5,3,3}$, $S_{4,4,3,5}$, $S_{4,4,3,4}$, $S_{4,4,3,3}$).

3.3.3 Algoritmo Exato - Sankoff com Modelo de Energia

As diferentes implementações do algoritmo de Sankoff utilizam variantes da mesma equação de recorrência [141]. Nesta seção, será apresentada uma variante proposta por Havgaard e Gorodkin [27] que implementa um modelo de energia mais complexo. As equações 3.9 e 3.10 apresentam um conjunto de equações que possuem diferentes escores representando situações como iniciar, continuar ou fechar um *loop*, fechar um *loop*, continuar uma *stack*. Além disso, são utilizadas duas matrizes: a matriz S, que contém escore das posições que estão pareadas e L que contém o escore das posições que não estão pareadas.

A Figura 3.12 apresenta uma representação gráfica desse conjunto de sequências. Nela, $S_{ij,kl}$ representa o escore da melhor estrutura e alinhamento quando os nucleotídeos na posição i, j e na posição k, l formam um par de base conforme ilustrado na parte superior da Figura 3.12. De forma similar, $L_{ij,kl}$ é o escore do melhor alinhamento onde as posições não formam um par de base. Os 6 primeiros termos da equação de recorrência são compostos pela soma de três tipos diferentes de escore: o primeiro tipo de escore (por exemplo, $S_{(i+1)(j-1),(k+1)(l-1)}$ ou $L_{(i+1)(j-1),(k+1)(l-1)}$) é formado pelo escore de um alinhamento já calculado. O segundo tipo de escore (o termo A) representa o custo da substituição de um nucleotídeo de uma sequência pelo da outra, sendo portanto os custos de *match* e

mismatch. O terceiro tipo de escore (como, $E^{loop\ open}$) significa o custo de adicionar uma nova estrutura.

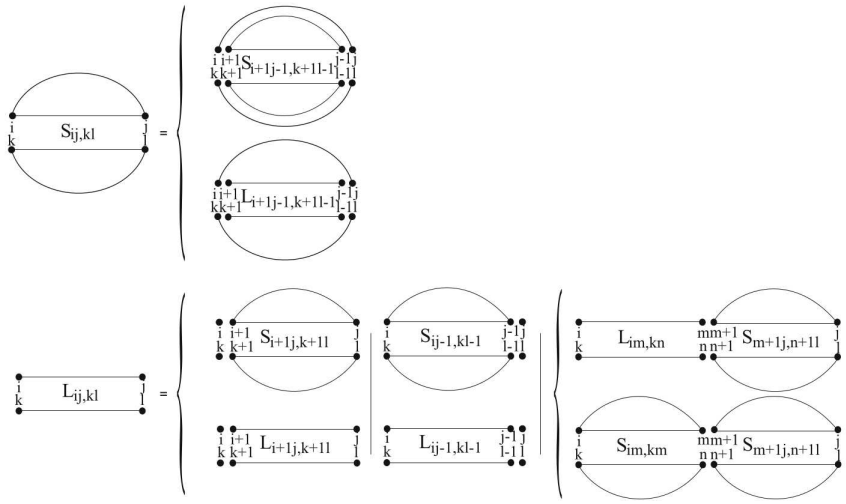


Figura 3.12: Representação gráfica das equações de recorrência do algoritmo de Sankoff [27].

$$S_{ij,kl} = \max \begin{cases} S_{(i+1)(j-1),(k+1)(l-1)} + A^{basepair}_{p_i p_j p_k p_l} \\ L_{(i+1)(j-1),(k+1)(l-1)} + A^{basepair}_{p_i p_j p_k p_l} + E^{stack}_{n_i n_j n_k n_l, n_{i+1} n_{j-1} n_{k+1} n_{l-1}} \end{cases} \tag{3.9}$$

$$L_{ij,kl} = \max \begin{cases} S_{(i+1)j,(k+1)l} + A^{single}_{n_i n_k} + E^{loop\ open}_{n_i n_k, n_{i+1} n_j n_{k+1} n_l} \\ L_{(i+1)j,(k+1)l} + A^{single}_{n_i n_k} + E^{loop\ extend} \\ S_{i(j-1),k(l-1)} + A^{single}_{n_j n_l} + E^{loop\ open}_{n_j n_l, n_i n_{j-1} n_k n_{l-1}} \\ L_{i(j-1),k(l-1)} + A^{single}_{n_j n_l} + E^{loop\ extend} \\ \max\limits_{i<m<j,k<n<l} \begin{cases} L_{imkn} + S_{m(+1)j,(n+1)l} + E^{bifurcation\ loop} \\ S_{im,kn} + S_{(m+1)j,(n+1)l} + E^{bifurcation\ loop} \end{cases} \end{cases} \tag{3.10}$$

As equações de recorrência 3.9 e 3.10 também possuem um termo responsável por combinar duas subestruturas de forma que combinadas possuem o maior escore. Desta forma, duas subestruturas são combinadas para formar o maior escore, e o último termo da equação é o custo de combinar duas dessas estruturas.

3.3.4 Algoritmo de Sankoff para Múltiplas Sequências

Teoricamente, é possível estender o algoritmo de Sankoff (Seção 3.3.2) para a solução do alinhamento múltiplo ótimo. Neste caso, a complexidade torna-se $O(n^{3k})$, onde k é o número de sequências e n é o tamanho delas e a complexidade de memória é $O(n^{2k})$.

Isso significa que até mesmo para o caso de três sequências o algoritmo de Sankoff torna-se muito custoso para sequências curtas. Apesar de teoricamente ser capaz de resolver o problema para qualquer número de sequências, é muito comum o uso de heurísticas até para duas sequências e, até onde sabemos, não existe implementação do algoritmo de Sankoff para mais de duas sequências.

3.3.5 Métodos Heurísticos

Conforme visto na Seção 3.3.2, o algoritmo de Sankoff possui uma alta complexidade computacional e por isso várias heurísticas são utilizadas para reduzir o tempo e memória. O desafio desses métodos consiste em reduzir o espaço de busca e manter a qualidade do alinhamento. Nesta seção iremos descrever 3 variantes heurísticas baseadas no algoritmo de Sankoff que obtém o alinhamento estrutural para pares de sequências: Foldalign, PMcomp e LocARNA. A partir delas, foram criados métodos heurísticos progressivos (Seção 2.3.3.1) que permitem o alinhamento para mais de duas sequências: FoldalignM, PMmulti e mLocARNA. Esses métodos também são descritos nesta seção.

3.3.5.1 Foldalign

Foldalign [26] é uma variante heurística do algoritmo de Sankoff (Seção 3.3.2) para duas sequências. Uma versão simplificada da recursão do Foldalign está ilustrada na Equação 3.11 e uma versão completa pode ser encontrada em [39].

O Foldalign 1.0 [26] é considerado por muitos como a primeira variante heurística do algoritmo de Sankoff. Para reduzir o consumo de tempo e memória requeridos, e são utilizadas as seguintes restrições: (a) o alinhamento final é limitado a λ nucleotídeos; e (b) o tamanho máximo da diferença entre duas subsequências sendo alinhadas é limitado à δ nucleotídeos. Essas duas restrições reduzem a complexidade de memória do algoritmo para $O(n^2\lambda^2\delta^2)$, onde n é o comprimento das sequências e a complexidade de espaço é reduzida para $O(\lambda^3\delta)$.

O Foldalign 2.0 [41], introduziu um modelo mais complexo de substituição, de melhor relevância biológica. A equação de recorrência do Foldalign está ilustrada na equação

3.11.

$$D_{i,j,k,l} = \max \begin{cases} D_{i+1,j-1,k+1,l-1} + E_{bp}(n_i, n_j, n_k, n_l, \sigma_{bp}) & \text{(a)} \\ D_{i+1,j-1,k,l} + E_{bpiI}(n_i, n_j, -, -, \sigma_{bpiI}) & \text{(b)} \\ D_{i,j,k+1,l-1} + E_{bpiK}(-, -, n_k, n_l, \sigma_{bpiK}) & \text{(c)} \\ D_{i+1,j,k+1,l} + E_{al}(n_i, n_k, \sigma_{al}) & \text{(d)} \\ D_{i,j-1,k,l-1} + E_{ar}(n_j, n_l, \sigma_{ar}) & \text{(e)} \\ D_{i+1,j,k,l} + E_{glI}(n_i, -, \sigma_{glI}) & \text{(f)} \\ D_{i,j-1,k,l} + E_{grI}(n_j, -, \sigma_{glrI}) & \text{(g)} \\ D_{i,j,k+1,l} + E_{glK}(-, n_k, \sigma_{glK}) & \text{(h)} \\ D_{i,j,k,l-1} + E_{grK}(-, n_l, \sigma_{grK}) & \text{(i)} \\ \max\limits_{\substack{i<m<j \\ k<n<l}} \{D_{i,m,k,n} + D_{m+1,j,n+1,l} + \sigma_{mbl}\} & \text{(j)} \end{cases} \tag{3.11}$$

Na equação 3.11, o caso (a) adiciona um par de bases em ambas as estruturas. Os casos (b) e (c) adicionam um par de bases em uma sequência e um *gap* na outra. Os casos (d) e (e) adicionam um nucleotídeo não pareado ao final do alinhamento. Os casos (f), (g), (h), e (i) adicionam um nucleotídeo não pareado em uma sequência e um *gap* na outra sequência. O caso (j) é a regra de *multibranch* onde duas subestruturas são combinadas para formar uma estrutura maior. Nesta equação de recorrência, os termos E são funções de escore baseadas na enegia σ da estrutura secundária em questão.

O Foldalign 2.1 [38] adicionou uma nova heurística, a constante de *pruning*. Neste método, um sub-alinhamento é eliminado quando um escore atinge um valor muito ruim pré-determinado. Isto significa que o escore do alinhamento é tão ruim, que provavelmente não faz parte de um alinhamento biologicamente relevante. Com essas limitações, ele não garante obter a solução ótima, e em alguns casos nenhuma solução é obtida.

Para implementar a heurística de *pruning*, o Foldalign 2.1 utiliza programação dinâmica *forward*. Desta forma, ao se processar uma célula que possui um escore inferior a uma constante, o algoritmo não aplica os membros da equação de recorrência a ela. Esse método é conhecido na literatura e é utilizado também nos programas MSA 1.0 e MSA 2.0 (Seção 2.3.1.2), porém em vez de se utilizar uma constante passada como argumento pelo usuário, eles utilizam o limite de Carrillo-Lipman (Seção 2.3.1.1), que ainda garante a otimalidade.

Para verificar a melhoria do *pruning*, um teste foi realizado comparando execuções do Foldalign sem a heurística e com a heurística. Nele foram usadas sequências reais de várias famílias de RNA (5S rRNA, Purine, THI U1, tRNA). Na comparação foi verificado que o Foldalgin com *pruning* localiza os mesmos resultados que a versão sem *pruning*,

porém executa 5 vezes mais rápido.

Para avaliar a capacidade de detectar alinhamentos, o Foldalign foi comparado com outras estratégias estado da arte: Dynalign [33], LocARNA [134], Consan [18] e Steamloc [50]. Neste experimento, foram utilizadas sequências reais, contendo tRNA e 5S rRNAs. As sequências foram separadas em conjuntos, de acordo com a similaridade entre elas. Para esse experimento, o Foldalign obteve a melhor performance para sequências com similaridade entre 10% e 60% e o Dynalign obteve melhor resultados para similaridade entre 65% e 100%.

3.3.5.2 PMcomp

Hofacker *et al.* propuseram uma variante do algoritmo de Sankoff chamada PMcomp [49]. Nela, assume-se que um modelo de estrutura para cada uma das sequências já é conhecido através do algoritmo de McCaskill (Seção 3.2.2).

Considere duas sequências S_1 e S_2, com suas respectivas matrizes de probabilidades P^{S_1} e P^{S_2}. Com isso, o PMcomp calcula Ψ_{ij}^{S} é o escore relativo à posição (i, j) na matriz de probabilidade da sequência S. O PMcomp permite duas formas de cálculo deste escore. Na primeira forma utiliza-se diretamente o escore da matriz de probabilidades, multiplicado por um número inteiro, que pode ser informado pelo usuário. A segunda forma utiliza o *log score*, dado pela equação 3.12:

$$\Psi_{ij} = \log(P_{ij}/p_{min}) \tag{3.12}$$

Nesta equação, P_{ij} é o valor da posição (i, j) da matriz de probabilidade P, como calculado pelo algoritmo de McCaskill (Seção 3.2.2) e p_{min} representa o valor da mínimo para se considerar uma probabilidade ser significante.

Portanto, para resolver o problema do alinhamento estrutural das sequências S_1 e S_2, considerando que o alinhamento possui N_{gap} e uma estrutura secundária E_1 e E_2, busca-se obter o maior escore possível com a fórmula 3.13.

$$\sum_{(ij;kl)\in E} (\Psi_{ij}^{S_1} + \Psi_{kl}^{S_2} + \tau(S_1(i), S_1(j), S_2(k), S_2(l))) + \gamma N_{gap} + \sum_{i\in S_1, k\in S_2 \notin E} \sigma(S_1(i), S_2(k)) \rightarrow \max \tag{3.13}$$

Na fórmula 3.13 o valor γ é a penalidade linear de *gaps*, a função $\sigma(S_1(i), S_2(k))$ descreve a substituição de nucleotídeos não pareados e a função $\tau(S_1(i), S_1(j), S_2(k), S_2(l))$ descreve o custo de substituição compensatória.

A obtenção do valor máximo é feita através de programação dinâmica. Considere $D_{i,j,k,l}$ é o melhor escore para as subsequências $S_1[i..j]$ e $S_2[k..l]$. Além disso, $D_{i,j,k,l}^{M}$ ser o melhor escore para os casos em que (i, j) e (k, l) sejam pares de bases. Desta forma, o

PMcomp apresenta duas equações de recorrência conforme 3.14 e 3.15

$$D_{i,j,k,l} = \max \begin{cases} D_{i+1,j,k,l} + \gamma \\ D_{i,j,k+1,l} + \gamma \\ D_{i+1,j,k+1,l} + \sigma(S_1(i), S_2(k)) \\ \max\limits_{\substack{m \leq j \\ n \leq l}} \{D^M_{i,m,k,n} + D_{m+1,j,n+1,l}\} \end{cases} \tag{3.14}$$

$$D^M_{i,j,k,l} = D_{i+1,j+1,k+1,l+1} + \Psi^{S_1}_{ij} + \Psi^{S_2}_{kl} + \tau(S_1(i), S_1(j), S_2(k), S_2(l)) \tag{3.15}$$

Na equação 3.14 os dois primeiros termos descrevem a adição de um *gap* em uma das sequências, o terceiro termo descreve a adição de nucleotídeos não pareados, o quarto termo descreve a junção de estruturas em uma maior para ambas as sequências e finalmente o último termo descreve a adição de nucleotídeo pareados.

O PMcomp calcula as equações 3.14 e 3.15 em tempo $O(n^6)$ e memória $O(n^4)$. Se considerarmos que $P^S_{ij} = 1$ para todo i, j se i e j formam um par de base, e $P^S_{ij} = 0$ caso contrário, o algoritmo é equivalente ao algoritmo de Sankoff [49].

3.3.5.3 LocARNA

Will *et al.* [134] propuseram o LocARNA (*local alignment of RNA*), um algoritmo para predição da estrutura secundária local em pares. No mesmo trabalho também foi apresentado o mLocARNA, um método para estender o LocARNA para múltiplas sequências. O LocARNA é implementado em C++, e seu desempenho permite o agrupamento de um grande grupo de ncRNAs.

Considere duas sequências S_1 e S_2, cada uma matrizes de probabilidades P^{S_1} e P^{S_2}, onde Ψ^S_{ij} é o escore relativo a posição (i, j) na matriz de probabilidade da sequência S. Assim como o PMcomp, o LocARNA tem a opção de utilizar o escore da matriz de probabilidades ou *log score*, porém no LocARNA ele é definido conforme a equação 3.16.

$$\Psi_{ij} = \begin{cases} \log \frac{P_{ij}}{p_0} / \log \frac{1}{p_0} & \text{Se } P_{ij} \geq p^* \\ -\infty & \text{Caso contrário.} \end{cases} \tag{3.16}$$

Nesta equação, P_{ij} é o valor da probabilidade na posição (i, j) como calculado pelo algoritmo de McCaskill (Seção 3.2.2), p^* é um valor de corte (*cutoff*), de forma que valores menores que ele sejam ignorados no cálculo do escore e p_0 é a probabilidade do pareamento ocorrer aleatoriamente. O termo $\log \frac{1}{p_0}$ representa uma normatização para os pesos serem ao máximo 1.

Para resolver este problema com programação dinâmica, considere $D_{i,j,k,l}$ como o escore máximo para o alinhamento das subsequências $S_1[i..j]$ e $S_2[k..l]$. A equação de recorrência está apresentada nas equações 3.17 e 3.18.

$$D_{i,j,k,l} = \max \begin{cases} D_{i,j-1,k,l} + \gamma \\ D_{i,j,k,l-1} + \gamma \\ D_{i,j-1,k,l-1} + \sigma(S_1(j), S_2(l)) \\ \max\limits_{j'l'} \{D_{i,j'-1,k,l'-1} + D^M_{j',j,l',l}\} \end{cases} \tag{3.17}$$

$$D^M_{i,j,k,l} = D_{i,j-1,k,l-1} + \Psi^{S_1}_{ij} + \Psi^{S_2}_{kl} \tag{3.18}$$

As células na matriz D^M são armazenadas apenas para nucleotídeos pareados. Porém, uma das heurísticas do LocARNA não armazena valores nas matrizes de programação dinâmica que não possuam um escore significante, conforme mostrado na equação 3.16, os valores menor que p^* são descartados. Uma variante desse conjunto de equações também é proposta, onde uma entrada 0 é adicionada na equação de recorrência para todos os casos que $i = k = 0$. São adicionados também escores negativos para *gaps* e *mismatches*. Normalmente, o *backtrace* é iniciado na última célula e termina na primeira célula. Nesta variante, ele é alterado para terminar em escores que possuam o valor 0. Assim, é possível remover prefixos das sequências alinhadas, calculando o alinhamento local.

3.3.5.4 FoldalignM

O FoldalignM [131] é um programa de alinhamento de estrutura secundária global múltiplo, baseado no Foldalign [38]. Ele constrói um alinhamento múltiplo progressivo computando todas as $\frac{N(N-1)}{2}$ comparações dos pares de sequências envolvidos, gerando uma matriz de probabilidade para cada comparação em pares. Então, alinha o par com o maior escore, calcula uma matriz de consenso e então alinha a próxima sequência a esta matriz de consenso e assim sucessivamente até que todas as sequências sejam alinhadas. O FoldalignM pode calcular a matriz de probabilidades usando o algoritmo de McCaskill (Seção 3.2.2) ou então usar a tabela de probabilidade do Foldalign para produzir uma árvore guia.

A escolha do agrupamento das sequências e escolha de qual sequência deve ser adicionada à matriz de probabilidade consenso é feita pelo método de *clustering*. Inicialmente, ordenam-se as sequências de acordo com seu escore. Então, são adicionadas as sequências com maior escore ao alinhamento do primeiro *cluster* e a lista de escores dos pares de sequência é percorrida de forma descendente. Caso o escore possua um valor de corte inferior a uma constante, então:

- Se nenhuma das sequências existir em um *cluster*, crie um novo *cluster* e adicione-o ambas as sequências a ele;
- Se uma das sequências existir em um *cluster*, adicione a outra sequência a este *cluster*;
- Se ambas as sequências já estiverem em um mesmo *cluster*, não faça nada;
- Se as sequências existirem em *clusters* diferentes, utilize o escore do alinhamento entre elas para ser o escore de diferença entre os dois *clusters*, ou faça a união dos *clusters* se o valor for inferior ao de corte.

O valor de corte p^* pode ser alterado pelo usuário. Este valor é essencial para a sensibilidade dos *clusters*, afinal um valor de corte alto irá causar muitas uniões (*merges*) das matrizes, criando poucos *clusters* de baixa similaridade. Por outro lado, valores baixos irão criar muitos *clusters*. Desta forma, este método de alinhamento é progressivo pois, inicialmente, é gerada a matriz de probabilidade entre as duas sequências mais próximas, sendo adicionadas as próximas sequências, de acordo com a similaridade, até que todas as sequências tenham sido adicionadas.

O programa FoldalignM [131] foi comparado a alguns outros programas heurísticos de alinhamento múltiplo. Nos testes foram utilizados conjuntos de 9 a 75 sequências RFAM classificados com baixa ou alta similaridade entre elas. A comparação mostra que o desempenho do programa é equivalente a outras abordagens, porém o FoldalignM se mostrou muito eficiente para sequências de baixa similaridade.

3.3.5.5 PMmulti

O PMmulti [49] foi proposto como uma ferramenta para o alinhamento múltiplo baseada no PMcomp (Seção 3.3.5.2). Dado um alinhamento estrutural do PMComp, é possível definir uma matriz de probabilidades consenso entre as duas sequências, conforme a equação 3.19.

$$P_{p,q}^{S_1 S_2} = \begin{cases} \sqrt{P_{i_p,j_q}^{S_1} P_{k_p,l_q}^{S_2}} & \text{para matches} \\ 0 & \text{para gaps} \end{cases} \tag{3.19}$$

Nesta equação, o termo i_p é a posição da sequência S_1 correspondente à posição p no alinhamento. Desta forma, é possível estender o método para a criação de alinhamento para mais de duas sequências.

Para o alinhamento de N sequências de RNA, calcula-se todos $N(N-1)/2$ alinhamentos em pares das sequências. Então o escore de similaridade dos alinhamentos é utilizado

para criar um árvore guia e usando a equação 3.19 o alinhamento múltiplo progressivo (Seção 2.3.3.1) das sequências é criado.

Em seus resultados, o PMmulti foi comparado com outras 3 ferramentas: ClustalW, PMstring e MARNA para o alinhamento de 7 sequências reais, cuja estrutura secundária é conhecida manualmente. Eles mostraram que, nesse experimento, o ClustalW produz um alinhamento de baixa qualidade, dado que o alinhamento é exclusivamente baseado na estrutura primária. O MARNA e o PMstring produzem um alinhamento aceitável, porém o PMmulti foi capaz de encontrar uma mutação adicional, sendo o método que mais se aproximou do alinhamento manual.

3.3.5.6 mLocARNA

O mLocARNA é o método de alinhamento múltiplo progressivo baseado nos alinhamentos em pares do LocARNA (Seção 3.3.5.3). Da mesma forma que o PMmulti faz com o PMcomp, ele cria o alinhamento em pares de todas as sequências e usa o escore de similaridade para criar um árvore guiada.

Porém, foi notado que o PMmulti utiliza um escore 0 na matriz de consenso para *gaps*. Consequentemente, quando existe um *gap* em uma das sequências, o escore na matriz de consenso é muito baixo, por isso o PMmulti tem uma tendência de remover muitos pares de bases para um grande número de sequências.

Para resolver isso, o mLocARNA define a matriz de probabilidades consenso diferentemente. Dada duas sequências, S_1 e S_2, considere que S é alguma dessas sequências:

$$P_{p,q}^{S_1 S_2} = \sqrt{\bar{P}_{p,q}^{S_1} \bar{P}_{p,q}^{S_2}}$$

$$\bar{P}_{p,q}^{S} = \begin{cases} max(p_0, P_{i_p,j_q}^{S}) & \text{para matches} \\ p_0 & \text{caso contrário} \end{cases} \tag{3.20}$$

onde i_p e i_q são as posições correspondente a p e q no alinhamento da sequência S e p_0 representa a probabilidade de pares de bases ocorrerem aleatoriamente. O restante do algoritmo não foi alterado, onde o escore dos alinhamento par-a-par são utilizados para criar uma árvore guiada e obter o alinhamento múltiplo ao final.

O mLocARNA foi testado em um conjunto de 3.901 sequências de 503 famílias. Seus resultados mostraram que a ferramenta é capaz de agrupar classes conhecidas de RNA, tal como o tRNAs, mas também agrupou alguns candidatos a novas classes de ncRNA. Em alguns casos, foi possível adicionar algumas sequências à famílias de RNA já conhecidas, e além disso, o mLocARNA detectou um homólogo do mir-126 e outro do mir-7 que não foram detectados em nenhum estudo computacional anterior.

3.4 Outras Características de Sequências de RNA

Além dos métodos apresentados, algumas outras características de sequências de RNAs são importantes para essa Tese e são descritas nessa seção. A Seção 3.4.1 descreve o conceito de GC-Content. A Seção 3.4.2 descreve um banco de dados de alinhamento de RNA. Finalmente, a Seção 3.4.3 apresenta um esquema de escore de substituições e compensação.

3.4.1 GC-Content

GC-Content é uma características de sequências biológicas, que estabelece a proporção dos nucleotídeos G-C em relação aos outros. Dada uma sequência de RNA que possui os nucleotídeos $\Sigma = \{A, U, G, C\}$, considere σ_x, como sendo a quantidade de nucleotídeos x na sequência S. Desta forma, o GC-Content de uma sequência é definido pela Equação 3.21:

$$GC - Content(S) = \frac{\sigma_C + \sigma_G}{\sigma_C + \sigma_G + \sigma_A + \sigma_U} \tag{3.21}$$

Na Bioinformática, alguns algoritmos implementados possuem melhores resultados de acordo com o GC-Content das sequências e, por isso, muitas vezes deve ser considerado na análise de resultados.

3.4.2 RFAM

O RFAM [60] é um banco de dados muito utilizado e atualmente é um dos maiores bancos de dados de alinhamento de RNA disponível [48]. O banco de dados é de acesso público e disponível no endereço `http://rfam.org`. O RFAM consiste em uma coleção de famílias de RNA representadas por um alinhamento múltiplo de sequências, um estrutura secundária consenso e um modelo de convariância. As famílias de RNA são adicionadas ao banco de dados por uma combinação de análise manual, análise baseada na literatura e *pipeline* de software personalizados. A primeira versão do RFAM 1.0 [29] contém 25 famílias de RNA e em sua última versão [60], o RFAM 13.0 existe um total de 2.687 famílias.

3.4.3 RIBOSUM85-60

Assim como a comparação primária, a comparação secundária pode associar um valor único à substituição de pares de bases, ou pode utilizar matrizes de substituição. A equação 3.8 utiliza uma função de substituição $\sigma(S_1(i), S_2(k))$, responsável pelos nucleotídeos

não pareados, e $\tau(S_1(i), S_1(j), S_2(k), S_2(l)))$, responsável pela substituição de nucleotídeos pareados. Neste caso, as matrizes de substituição são duas matrizes de tamanho 4x4 e 16x16.

Klein e Eddy [62] publicaram uma das mais conhecidas matrizes de substituição para RNA, o RIBOSUM85-60. Os dados foram obtidos através de modelos estatísticos utilizando três bancos de dados de RNA com alinhamentos confiáveis. As Tabelas 3.1 e 3.2 ilustram respectivamente as matrizes de substituição (4x4) e compensação (16x16) RIBOSUM85-60.

Tabela 3.1: Matriz de substituição RIBOSUM85-60 [62].

A	2,22			
C	-1,86	1,16		
G	-1,46	-2,48	1,03	
U	-1,39	-1,05	-1,74	1,65
	A	C	G	U

Tabela 3.2: Matriz de compensação RIBOSUM85-60 [62].

AA	-2,49															
AC	-7,04	-2,11														
AG	-8,24	-8,89	-0,80													
AU	-4,32	-2,04	-5,13	4,49												
CA	-8,84	-9,37	-10,41	-5,56	-5,13											
CC	-14,37	-9,08	-14,53	-6,71	-10,45	-3,59										
CG	-4,68	-5,86	-4,57	1,67	-3,57	-5,71	5,36									
CU	-12,64	-10,45	-10,14	-5,17	-8,49	-5,77	-4,96	-2,28								
GA	-6,86	-9,73	-8,61	-5,33	-7,98	-12,43	-6,00	-7,71	-1,05							
GC	-5,03	-3,81	-5,77	2,70	-5,95	-3,70	2,11	-5,84	-4,88	5,62						
GG	-8,39	-11,05	-5,38	-5,61	-11,36	-12,58	-4,66	-13,69	-8,67	-4,13	-1,98					
GU	-5,84	-4,72	-6,60	0,59	-7,93	-7,88	-0,27	-5,61	-6,10	1,21	-5,77	3,47				
UA	-4,01	-5,33	-5,43	1,61	-2,42	-6,88	2,75	-4,72	-5,85	1,60	-5,75	-0,57	4,97			
UC	-11,32	-8,67	-8,87	-4,81	-7,08	-7,40	-4,91	-3,83	-6,63	-4,49	-12,01	-5,30	-2,98	-3,21		
UG	-6,16	-6,93	-5,94	-0,51	-5,63	-8,41	1,32	-7,36	-7,55	-0,08	-4,27	-2,09	1,14	-4,76	3,36	
UU	-9,05	-7,83	-11,07	-2,98	-8,39	-5,41	-3,67	-5,21	-11,54	-3,90	-10,79	-4,45	-3,39	-5,97	-4,28	-0,02
	AA	AC	AG	AU	CA	CC	CG	CU	GA	GC	GG	GU	UA	UC	UG	UU

Capítulo 4

Arquiteturas Paralelas e Computação de Alto Desempenho

A computação de alto desempenho é uma subárea da Ciência da Computação que teve seu início na década de 1960, quando pela primeira vez cogitou-se quebrar um problema em subproblemas menores, com o objetivo de resolvê-los em diversos elementos de processamento simultaneamente [17].

Na década de 1970, surgiram os supercomputadores vetoriais, que eram projetados para calcular rapidamente os elementos de um vetor ou matriz. Essas arquiteturas eram exemplos da categoria SIMD (*Single Instruction Multiple Data*) de Flynn [23] onde uma mesma instrução era executada ao mesmo tempo sobre diversos dados diferentes. Os supercomputadores vetoriais conseguiram ser uma ordem de magnitude mais rápidos do que as arquiteturas convencionais da época.

No início da década de 1980, tornou-se popular uma outra categoria de máquinas, denominadas SMP (*Symmetric Multiprocessor*), que eram exemplos da categoria MIMD (*Multiple Instruction Multiple Data*) de Flynn, onde códigos diferentes podiam ser executados ao mesmo tempo em processadores distintos. Essas máquinas geralmente utilizavam o paradigma de memória compartilhada para a troca de dados entre os processadores.

No final da década de 1980, ficou claro que as máquinas SMP não eram escaláveis, ou seja, não era possível conectar um grande número de processadores às mesmas, pois os meios de comunicação (barramentos) rapidamente se tornavam saturados [17].

Para solucionar esse problema, foram criados sistemas de computação distribuída, onde computadores relativamente completos eram conectados através de uma rede de interconexão, de modo a criar um sistema integrado. Nos sistemas de computação distribuída, a troca de dados se dava através do paradigma de troca de mensagens.

Nessa categoria, inserem-se os *clusters* de computadores, que foram definidos como sendo sistemas de computação distribuída que utilizavam *hardware* e *software* disponíveis

no mercado (*commodity components*) [106]. Os *clusters* de computadores rapidamente se disseminaram por possuírem um grande apelo comercial, que combinava baixo custo e relativa facilidade de programação.

Paralelamente à evolução dos *clusters*, arquiteturas específicas se disseminaram. Essas arquiteturas são geralmente otimizadas para execução de alguns tipos específicos de aplicações, sendo chamadas de aceleradores. Exemplos típicos de aceleradores são FPGAs (*Field Programmable Gate Arrays*), Intel Xeon Phi e GPUs (*Graphics Processing Units*).

Atualmente, observa-se uma integração entre essas diversas categorias de máquinas. Por exemplo, é comum a existência de *clusters* de *multicores* (SMPs) onde um ou mais *multicores* estão conectados a aceleradores (GPU, Intel Xeon Phi, FPGA ou outros).

Nas Seções 4.1, 4.2 e 4.3 são apresentadas as principais características das arquiteturas *multicore* e *clusters* de computadores, FPGAs e dos Intel Xeon Phi. Finalmente, na seção 4.4 as GPUs são apresentadas em mais detalhes.

4.1 Arquiteturas *Multicore*

A indústria de computadores busca constantemente evoluir o poder computacional de seus processadores. Anteriormente, a solução para aumentar o poder de processamento foi aumentar sistematicamente a frequência do *clock* das CPUs [127]. Para isso, foram desenvolvidas tecnologias que utilizam transistores cada vez menores. Porém, ao se aumentar a velocidade das CPUs, é produzido mais calor e com a redução do tamanho é mais difícil dissipar esse calor. Por volta de 2005, foi atingido um limite na frequência do relógio, a partir do qual o projeto do *chip* fica muito caro, devido ao tratamento do calor produzido.

Uma estratégia para aumentar a capacidade de processamento dos computadores é através da adição de diversas CPUs em um único computador. Um sistema multiprocessado é um sistema de computador no qual duas ou mais CPUs compartilham acesso total a uma memória RAM comum. Hoje, a maioria dos sistemas multiprocessados utilizam a arquitetura SMP (*Symmetric Multiprocessor*), onde existe uma cópia do sistema operacional na memória e qualquer CPU pode ser utilizada para executar o SO ou processos dos usuários.

Atualmente, existe uma distinção entre o *core* real e os *cores* vistos pelo SO devido à tecnologia *Hyper-Threading* (HT), permitindo que o SO escalone *threads* e processos em um *core* virtual como ele faria em um sistema multiprocessado convencional.

Inicialmente, os sistemas UNIX não manipulavam *threads*. Quando essa característica foi adicionada, surgiram diversos pacotes de *threads* em modo usuário. Porém, a existência de tantos pacotes dificulta a escrita de códigos portáteis. Para resolver esse problema,

foram criados alguns padrões para o gerenciamento de *threads*. Nesta Seção descrevemos 3 pacotes/ambientes de programação utilizados nesta Tese: POSIX *Threads* (Pthreads) (Seção 4.1.1), Open Multi-Processing (OpenMP) (Seção 4.1.2) e o C++11 (Seção 4.1.3).

4.1.1 POSIX Threads (Pthreads)

O pacote POSIX *Threads* (Pthreads) foi uma das primeiras padronizações de *threads* e foi criado como parte do POSIX. O Pthreads define um conjunto de funções, mas não define como elas devem ser implementadas.

O pacote pthread define funções para a criação de *threads* (pthread_create), término da *thread* (pthread_exit), espera pelo término de uma *thread* (pthread_join). Também oferece algumas funções para sincronização entre as *threads*, incluindo a criação e uso de *locks mutex* (pthread_mutex_init, pthread_mutex_destroy, pthread_mutex_lock, pthread_mutex_unlock) e uso de variável de condição (pthread_cond_init, pthread_cond_destroy, pthread_cond_wait e pthread_cond_signal).

O Algoritmo 4 ilustra um exemplo onde 5 *threads* são criadas e cada uma imprime uma mensagem na tela, junto com um número único identificador de cada *thread*. As linhas 6 a 8 definem uma estrutura utilizada para passar argumentos para as *threads*. Neste exemplo, a estrutura possui apenas um campo responsável por armazenar o identificador (linha 7). Cada *thread* executa a função "thread_func" (linhas 10 a 15). No pthread, as funções das *threads* sempre recebem apenas um argumento do tipo "(void*)". Dentro da função "thread_func", esse argumento é convertido para a estrutura definida anteriormente (linha 12). Então a *thread* imprime a mensagem na tela (linha 13) e retorna NULL, terminando a execução da *thread* (linha 14).

Na função "main" (linha 17), a thread principal define variáveis para armazenar os argumentos de cada *thread* (linha 20) e as infomações das *threads* criadas (linha 21). Um *loop* "for" é repetido NUM_THREADS vezes (linha 23). A cada iteração, a estrutura com os argumentos é inicializada (linha 25) e então a *thread* é criada (linha 26). Das linhas 27 a 31 são verificados erros na criação das *threads*. Depois disso, um outro *loop* "for" é repetido NUM_THREADS vezes (linha 34). A cada iteração, a *thread* principal aguarda o término de cada *thread* criada anteriormente (linha 36). Após isso, o programa termina a execução (linha 43).

4.1.2 OpenMP

O *Open Multi-Processing* (OpenMP) [102] é uma especificação de diversas diretivas de compilador e funções que permitem explorar o paralelismo em C, C++ e Fortran. O OpenMP permite expressar o paralelismo da aplicação nos *loops* do código-fonte, adicio-

Algoritmo 4 Exemplo “Hello, world!” usando Pthreads

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS     5

struct thread_arg {
    int id;
};

void *thread_func(void *arg)
{
    struct thread_arg *p = arg;
    printf("Hello, world! Thread %d\n", p->id);
    return NULL;
}

int main(void)
{
    int ret, i;
    struct thread_arg args[NUM_THREADS];
    pthread_t threads[NUM_THREADS];

    for (i = 0; i < NUM_THREADS; i++)
    {
        args[i].id = i;
        ret = pthread_create(&threads[i], NULL, &thread_func, &args[i]);
        if (ret)
        {
            perror("pthread_create");
            return -1;
        }
    }

    for (i = 0; i < NUM_THREADS; i++)
    {
        ret = pthread_join(threads[i], NULL);
        if (ret)
        {
            perror("pthread_join");
            return -1;
        }
    }
    return 0;
}
```

nando diretivas de compilação. Essas diretivas indicam como o trabalho e dados devem ser compartilhados entre as *threads* que serão executadas. Além disso, versões mais recentes do OpenMP permitem o tratamento de paralelismo em nível de tarefa.

O OpenMP é um padrão baseado no modelo de memória compartilhada. Ele define uma série de diretivas para definir como os dados serão compartilhados entre as *threads* (shared/private). Define formas de sincronização entre as *threads*, com diretivas que garantem que não irá executar simultaneamente (critical) ou criar uma barreira de sincronização entre as *threads* (barrier). Também define outras diretivas para as políticas de escolonamento (static/dynamic), entre outros.

Algoritmo 5 Exemplo "Hello, world!" usando OpenMP

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  #define NUM_THREADS     5
5
6  int main(void)
7  {
8      int i;
9
10     #pragma omp parallel for num_threads(NUM_THREADS) private(i) schedule(dynamic)
11     for (i = 0; i < NUM_THREADS; i++)
12     {
13         printf("Hello, world! Thread %d\n", i);
14     }
15     return 0;
16 }
```

O Algoritmo 5 ilustra um exemplo de criação de 5 *threads*, onde cada uma imprime uma mensagem na tela com um número único identificador. Na função principal (linha 6) foi definida a execução do *loop* for em paralelo (linha 10), com as seguintes propriedades: são criadas "NUM_THREADS" *threads*, cada uma possui a variável *i* privada, cada *thread* é escalonada com a política de escalonamento dinâmica, cada *thread* executa uma iteração do *loop* e ao terminar a execução recebe uma nova iteração. Durante a execução das *threads*, cada uma imprime uma mensagem com seu número único na linha 13.

4.1.3 Threads em C++11

O padrão C++11 adicionou à linguagem C++ a capacidade de criar e gerenciar *threads* [84]. Foram adicionados diversos *headers* à biblioteca padrão para auxiliar a criação e sincronização entre as *threads*. Podemos citar entre eles: "<threads>" contém as informações sobre criação, término e espera pelo término das *threads*, "<condition_variable>" para a criação de varíaveis de condição, "<mutex>" para a criação e uso de *locks mutexes*, "<atomic>" para o uso de operações atômicas, entre outros.

Algoritmo 6 Exemplo "Hello, world!" usando *threads* em C++11

```
1  #include <stdio.h>
2  #include <thread>
3
4  #define NUM_THREADS     5
5
6  void *thread_func(int i)
7  {
8      printf("Hello, world! Thread %d\n", i);
9      return NULL;
10 }
11
12 int main(void)
13 {
14     int i;
15     std::thread threads[NUM_THREADS];
16
17     for (i = 0; i < NUM_THREADS; ++i)
18     {
19         threads[i] = std::thread(thread_func, i);
20     }
21
22     for (i = 0; i < NUM_THREADS; ++i)
23     {
24         threads[i].join();
25     }
26     return 0;
27 }
```

O Algoritmo 6 ilustra o exemplo de criação de 5 *threads* que imprimem uma mensagem na tela com um número identificador. A função executada por cada uma das *threads* é definida das linhas 6 a 10. Essa função recebe como argumento um identificador único através de um inteiro (linha 6), imprime a mensagem na tela (linha 8) e retorna NULL, finalizando a *thread* na linha 9.

Na função principal (linhas 12 a 27), são reservadas variáveis para armazenar informação de cada uma das *threads* (linha 15), Um *loop* "for" é repetido NUM_THREADS vezes (linha 17), a cada iteração uma nova *thread* é criada (linha 19). Depois da criação das *threads*, um novo *loop* "for" é repetido NUM_THREADS vezes (linha 22), desta vez a *thread* principal aguarda o término de cada uma das *threads* criadas anteriormente (linha 24). Na linha 26 a *thread* principal termina a execução.

4.1.4 *Clusters* de Computadores

Os *clusters* de computadores são um conjunto de computadores conectados através de *hardware* e *software* especializado, apresentando uma imagem única de sistema (*Single System Image*) aos usuários [132]. A imagem única de sistema permite que seja criada a ilusão de que os vários computadores que compõem o *cluster* são na verdade um único sistema.

Em um *cluster*, cada nó de processamento é um computador praticamente completo, com processador, memória e disco rígido, geralmente sem monitor, mouse ou teclado [53].

Os nós de processamento são interligados por uma rede de interconexão, geralmente de alta velocidade e disponível no mercado. Como vantagens do *cluster*, podemos citar:

- Potencial para escalabilidade, alta disponibilidade e alto desempenho;
- Relativa facilidade em adicionar/remover componentes;
- Relativa facilidade de programação, entre outros.

Atualmente, os *clusters* são geralmente programados com o padrão MPI [3] (*Message Passing Interface*), que permite a troca de mensagens síncronas e assíncronas entre os processos. Além disso, oferece primitivas de *broadcast* "*one to all*" e "*all to all*", entre outras.

Apesar dos *clusters* SMPs (ou *clusters* de *multicores*) poderem ser programados unicamente com troca de mensagem, uma maneira mais eficiente de utilizá-los consiste em se usar um modelo de programação híbrido, onde os núcleos trocam valores por memória compartilhada, com Pthreads (Seção 4.1.1) ou OpenMP (Seção 4.1.2), e os computadores trocam mensagens com MPI [30].

4.2 Field Programmable Gate Arrays

Os *Field Programmable Gate Arrays* (FPGAs) [135] são circuitos integrados em larga escala que podem ter sua configuração interna modificada após a sua fabricação. Uma de suas características mais importantes são baixo o consumo de energia e um potencial alto *throughput* de operações. A capacidade do FPGA é medida pelo número de blocos lógicos configuráveis que ele possui ou pelo número de portas lógicas equivalentes.

Quase todos os FPGAs possuem blocos programáveis compostos. Esses blocos são compostos por registradores, elementos lógicos configuráveis e interconexões. Além disso existem outras características comuns aos FPGAs [135]:

- Elementos de Computação: compostos por alguns registradores e elementos lógicos de baixo nível.
- Tabelas de *Lookup*: Também chamadas de *Lookup Table* (LUT). Geralmente estão associadas a um ou mais registradores programáveis e podem implementar funções lógicas através de diferentes entradas. Os detalhes de implementação variam de acordo com a família e o fabricante.
- Memória: A memória pode ser global para todos elementos lógicos ou local para um determinado grupo de elementos.

- Recursos de Roteamento: Fazem a interligação entre os elementos de processamento, memória interna e outras estruturas do circuito. Possuem velocidades e níveis de flexibilidade bem diferentes.
- Entrada e Saída Configurável: A placa de FPGA deve se comunicar com um computador hospedeiro. Normalmente possuem circuitos chamados de blocos de entrada e saída cuja função é fazer o gerenciamento da comunicação.

Os FPGAs são geralmente programados com linguagens de descrição de *hardware* (*Hardware Description Languages* - HDL) que, ao serem compiladas, geram descrições de circuitos que são descarregadas no FPGA, para execução. As linguagens de descrição mais comuns são VHDL (VHSIC Hardware Description Language) e Verilog. Mais recentemente, alguns esforços foram feitos para permitir a execução de código OpenCL (Open Computing Language) em FPGAs [114]. Apesar de não ser uma linguagem específica para desenho de circuitos, essa abordagem possui como vantagem a facilidade em criar uma estratégias heterogêneas combinando CPU e FPGA.

4.3 Intel Xeon Phi

A arquitetura *Many-Integrated Core* (MIC) foi lançada pela Intel em 2013, é uma arquitetura que combina uma quantidade maior de núcleos quando comparada às CPUs tradicionais. Essas placas aceleradoras possuem como características [58]:

- Um novo componente da arquitetura MIC é a *vector processing unit* (VPU). Cada VPU está associada a um núcleo que executa instruções SIMD sobre registradores de 512 bits, podendo executar simultaneamente 16 operações de ponto flutuante ou 8 operações de ponto flutuante com precisão dupla.
- Comparada aos processadores tradicionais (Intel Xeon), a arquitetura Intel MIC tem um número maior de *cores* de tamanho reduzido, além de ter mais *threads* de *hardware* e unidades vetoriais maiores.
- O Intel Xeon Phi possui diferentes abordagens para a execução de aplicações:
  - Nativo: quando o binário é transferido e executado diretamente no acelerador. Isso é possível pois o Intel Xeon Phi possui seu próprio sistema operacional, chamado de μOS, baseado em GNU/Linux e permite o acesso por conexão ssh.
  - *Offload*: quando uma aplicação é iniciada no *host*, mas o programador marca frações da aplicação para serem executadas no acelerador.

Knights series é o codinome dado para uma geração de produtos da Intel com a arquitetura MIC. As gerações são: *Knights Ferry*, a primeira geração usada como teste e não foi lançada comercial; *Knights Corner*, a primeira geração disponível para o público em geral e produzida com tecnologia de transistores de 22 nm [55]; *Knights Landing*, a segunda geração disponível comercialmente, que difere da geração anterior por utilizar tecnologia de 14 nm [56], aumentar a eficiência de energia por core, incluir o conjunto de instruções AVC-512, além de permitir arquitetura de memória DDR e MCDRAM [117].

Os Intel Xeon Phi são geralmente programados com a linguagem C associada a algumas extensões como OpenMP, MPI e Pthreads. Frequentemente são utilizados os compiladores da Intel, otimizados pelo fabricante para para detectar automaticamente trechos de código em que é possível utilizar as instruções SIMD para a arquitetura MIC, além de possuir várias opções específicas para a arquitetura, como por exemplo a escolha do modo de operação (Nativo ou *Offload*).

4.4 Unidades de Processamento Gráfico

As Unidades de Processamento Gráfico (*Graphics Processing Unit* (GPU)) são microprocessadores com funções otimizadas para acelerar a renderização gráfica, executando em *hardware* as funções matemáticas mais comuns.

Na metade da década de 1990, iniciou-se uma forte demanda dos consumidores por aplicações que empregavam gráficos 3D, principalmente na área de entretenimento eletrônico. As placas gráficas tornaram-se populares e progressivamente foram aumentando a complexidade e criando ambientes tridimensionais mais realistas. Nesta época três empresas, NVidia, ATI e 3dfx, começaram a lançar aceleradores gráficos a preços acessíveis, que se tornaram aceleradores gráficos populares [110].

As GPUs possuem uma capacidade de executar paralelamente a mesma operação sobre dados diferentes, conceito chamado de SIMD (*Single Instruction Multiple Data*) na taxonomia de Flynn [23]. O alto *throughput* de operações aritméticas chamou a atenção de desenvolvedores e pesquisadores que desejavam acelerar alguns problemas utilizando tal arquitetura.

Do ponto de vista da programação paralela, um marco importante nas tecnologias dos aceleradores gráficos ocorreu no ano de 2001. Neste ano foi lançado o primeiro chip industrial (NVidia GeForce 3) que implementava o recém-criado padrão da Microsoft DirectX 8.0, que exigia que o *hardware* tivesse um *vertex* programável e um estágio de *pixel shading* programável. Pela primeira vez, desenvolvedores tinham acesso a exatamente quais funções seriam executadas nas GPUs de suas estações de trabalho [110].

Ao final da década de 1990, eram muito limitadas as formas de interação com as GPUs. Nessa época, todo programa deveria simular uma renderização tradicional de vídeo para poder executar em uma GPU. Desta forma, era muito complicado realizar operações com ponto flutuante, muito desejadas por aplicações científicas, e operações de *debug* eram também muito difíceis.

De maneira a simplificar a programação das GPUs e permitir seu uso em áreas que não fossem o processamento de imagens, surgiu o conceito de programação de propósito geral em GPUs (*General-purpose computing on graphics processing units* - GPGPU), que é o uso ambientes e ferramentas de programação paralela para executar códigos de propósito geral em unidades de processamento gráfico.

Algumas arquiteturas foram criadas para facilitar o desenvolvimento de programas de propósito geral para as GPUs, dentre as quais podemos citar CUDA [96], que será detalhada na Seção 4.4.1.

4.4.1 Arquitetura CUDA

A arquitetura *Compute Unified Device Architecture* (CUDA) foi proposta pela NVidia em novembro 2006 [110]. É composta por elementos em *hardware* de alto desempenho e *software*. A Figura 4.1 ilustra os elementos em *software* da arquitetura.

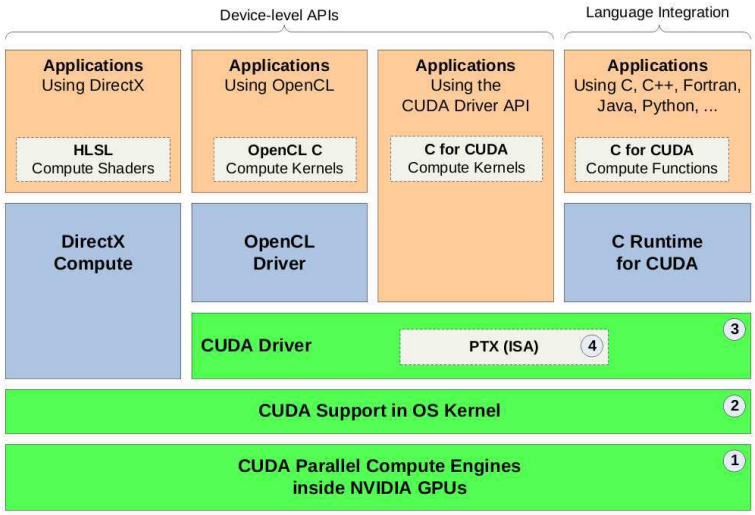


Figura 4.1: Software utilizado na arquitetura CUDA [95].

Na camada mais baixa da arquitetura CUDA, existem *engines* paralelas de computação que ficam presentes dentro das placas NVidia. Na camada imediatamente superior existe um *driver* do sistema operacional que é responsável pela inicialização do *hardware*, além da configuração e comunicação com a GPU. Acima desta camada, está um outro

driver (CUDA Driver) que se executa em modo de usuário e provê uma interface de programação (API) de baixo nível. Dentro dessa API, existe o *Parallel Thread eXecution* (PTX) que define uma máquina virtual de baixo nível e um conjunto de instruções da arquitetura (ISA) para execução de *threads* paralelas de propósito geral [92]. Compiladores de alto nível como o CUDA geram instruções PTX que são otimizadas e traduzidas para um conjunto de instruções específicas da arquitetura disponível.

O nível acima da arquitetura ilustra as diversas possibilidades que estão disponíveis para se programar a GPU, além de API CUDA: uma camada OpenCL [101], *framework* desenvolvido para a execução de aplicações paralelas em ambientes heterogêneos e algumas camadas para suporte CUDA em outras linguagens, como C, Fortran, Python.

O ambiente de desenvolvimento de software provê um conjunto de ferramentas úteis para desenvolver aplicações na arquitetura CUDA. Podemos citar as bibliotecas que provêm soluções otimizadas para a arquitetura, como a transformada rápida de Fourrier, *C Runtime for CUDA*, que permite a executar funções padrão do C, além de ferramentas de compilação, *debug* e *Visual Profiler*.

4.4.2 Modelo de Programação em CUDA

Um programa em CUDA possui a capacidade de executar instruções na GPU. Ele pode ser dividido em duas partes: o código "*host*" que é executado na CPU, e o código "*device*", que se executa na placa de vídeo. Os *kernels* são a forma pela qual o código

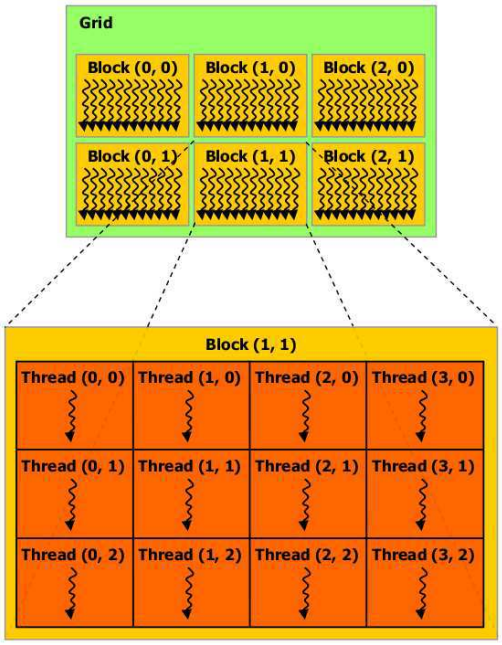


Figura 4.2: Modelo de programação CUDA [96].

host é capaz de chamar o código *device*, sendo assim possível a interação entre eles. Toda função executada na placa gráfica está dentro de um *kernel*.

O modelo de programação CUDA permite que várias instâncias sejam executadas em paralelo por diferentes *threads*. Cada *thread* possui um conjunto de registradores e uma memória local privada. Além disso, cada *thread* possui um identificador único dentro do seu bloco, que pode ser acessado através da variável local *built-in* $threadIdx$.

Conforme ilustrado na Figura 4.2, as *threads* podem ser agrupadas em blocos. *Threads* que estão no mesmo bloco podem se comunicar através de um tipo mais rápido de memória, e podem ser sincronizadas através de uma diretiva de barreira ($\_\_syncthreads()$). Os blocos de *threads* podem possuir de uma a três dimensões.

Vários fatores influenciam a ordem na qual os blocos são executados, como o número de registradores que cada *thread* utiliza, e a quantidade de recursos [96]. Os blocos são escalonados em vários multiprocessadores. À medida que a execução de um bloco é terminada, um novo bloco é alocado àquele processador até que todos os blocos tenham sido executados. Porém, a ordem de execução dos blocos não pode ser determinada pelo programador. Sendo assim, é necessário que os blocos não possuam uma dependência de dados entre si.

Os blocos de *threads* são organizados *grids*. Em versões mais recentes do CUDA é possível criar *grids* de uma a três dimensões, assim como os blocos. Cada bloco possui um identificador único que pode ser acessado pelo *kernel* através da variável $blockIdx$. Ao contrário dos blocos, não existem chamadas de sincronia entre os *grids*. É esperado que a sincronia dos *grids* sejam feitas através de diferentes chamadas de *kernels*.

A Figura 4.3, ilustra a arquitetura de memória em CUDA, os níveis de agrupamento das *threads* e a memória relacionada. Os tipos de memória disponíveis na arquitetura são:

- **Local:** memória que apenas a *thread* pode acessar.
- **Compartilhada:** memória otimizada para uma rápida comunicação entre as *threads*. Permite a comunicação apenas entre *threads* do mesmo bloco.
- **Global:** memória que pode ser lida e escrita por todas as *threads*.
- **Textura:** permite acelerar o acesso a uma memória global de leitura (*read-only*). Para que este acesso seja acelerado, é necessário que as *threads* possuam algum padrão comum de acesso a memória (*coalesced*) [96].
- **Constante:** pequena memória rápida que pode ser lida por todas as *threads* (*read-only*).

O Algoritmo 7 ilustra o exemplo de um "Hello, world!" em CUDA C. A função printf utilizada em *kernels* está disponível apenas para GPUs com compute capability 2.0 ou

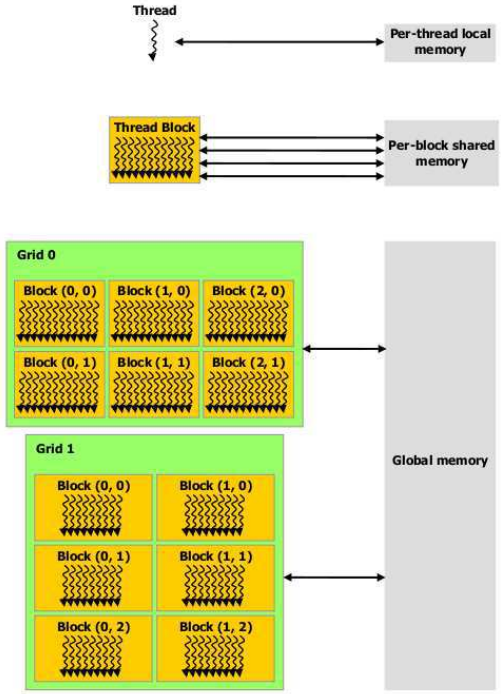


Figura 4.3: Hierarquia da memória em CUDA [96].

Algoritmo 7 Exemplo "Hello, world!" usando CUDA

```
1  #include <stdio.h>
2
3  __global__ void helloKernel()
4  {
5      printf("Hello, world! Thread %d\n", threadIdx.x);
6  }
7
8  int main(int argc, char **argv)
9  {
10     dim3 numBlocks(1);
11     dim3 numThreads(5);
12
13     helloKernel<<<numBlocks, numThreads>>>();
14     cudaDeviceSynchronize();
15     return 0;
16 }
```

maior. No Algoritmo 7, o kernel "helloKernel" é definido das linhas 3 a 6. Nele, cada *thread* imprime uma mensagem na tela, junto com um número único identificador na linha 5. Na função principal (linha 8), são inicializadas variáveis que contém o número de blocos (linha 10) e *threads* por bloco (linha 11). Então, o *kernel* é chamado na linha 13. Na linha 14, a função "cudaDeviceSynchronize()" garante que a tarefa executando em GPU terminará antes da execução prosseguir e o programa termina na linha 15.

4.4.3 Arquitetura Kepler

A arquitetura Kepler [93] foi apresentada pela NVidia em abril de 2012 e está disponível em diversas placas deste fabricante. Ela foi lançada para substituir a série GF 100, com arquitetura Fermi.

Cada GPU da NVidia é composta por vários *Streaming Multiprocessor* (SM), que são as menores unidades de execução da arquitetura. Um SM é um conjunto CUDA *cores* de precisão simples, agrupados com outros elementos, como *cores* de precisão dupla, memória compartilhada, cache L1 e o *Special Function Units* (SFUs), que são unidades responsáveis por processamento de operações matemáticas mais complexas. Nas placas com arquitetura Kepler, até 192 CUDA *cores* de precisão simples, 64 CUDA *cores* de precisão dupla, 32 SFUs e 32 unidades de *load/store* (LD/ST), são agrupados para formar 1 SM [93], como pode ser visto na Figura 4.4.

A Figura 4.5 representa a arquitetura Kepler, onde 15 SMs são posicionados ao redor

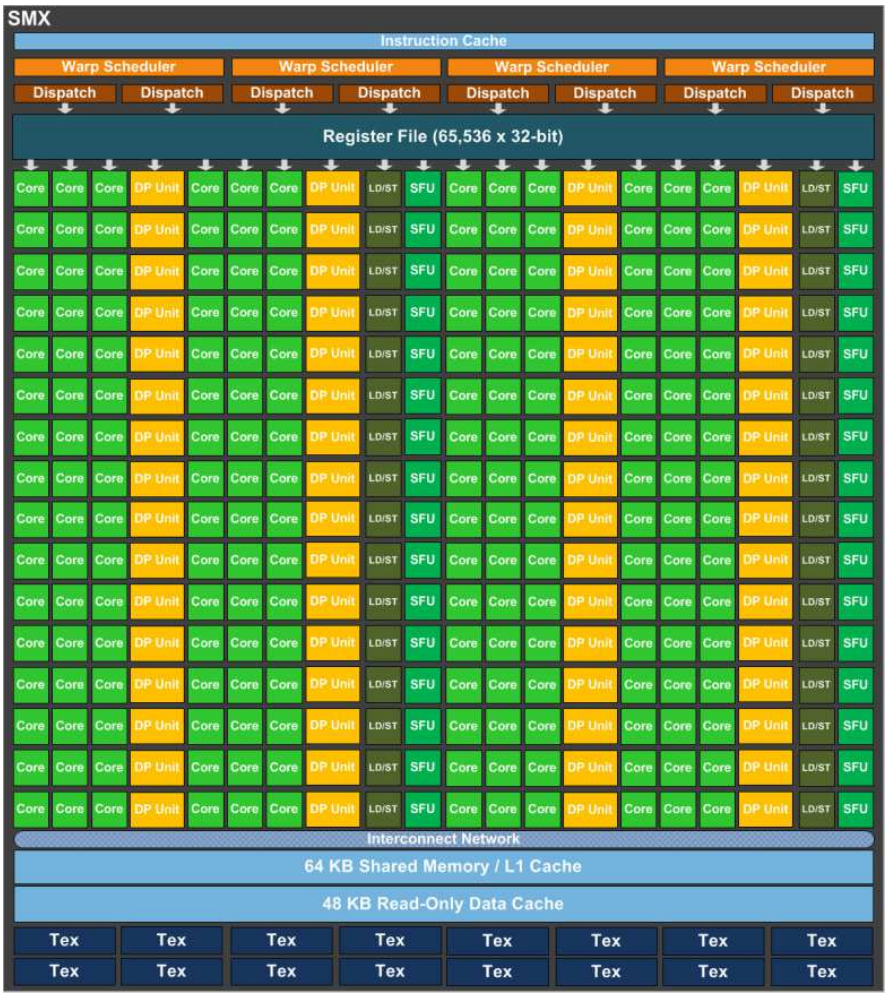


Figura 4.4: SM da arquitetura Kepler [93].

Figura 4.5: Diagrama da arquitetura Kepler com 15 SMs. Alguns modelos da família podem apresentar apenas 13 ou 14 SMs [93].

de um cache L2 comum. O cache é representado na figura pela cor mais clara e central. As porções médias nos extremos da figura representam as unidades de escalonamento (*warp scheduler* e *dispatch unit*) em conjunto com um cache L1.

A arquitetura Kepler possui uma série de mudanças em relação à sua predecessora, a arquitetura Fermi, dentre elas [93]: uma nova arquitetura de SM, melhorias no subsistema de memória, oferecendo melhorias no cache, maior largura de banda, todas as placas da arquitetura utilizam transistores de 28 nm e possuem o dobro do número de registradores 32-bit (65.536). As placas Kepler possuem um CUDA *Compute Capability* 3.0 ou 3.5, de acordo com o modelo, que permite um aumento na quantidade máxima de *threads* por multiprocessador (2048), além de possuir o dobro no número de blocos de *threads* (16).

Além disso, algumas versões mais novas da arquitetura adicionaram novas funcionalidades, como o paralelismo dinâmico (disponível em GPUs com *Compute Capability* 3.5 ou superior). O paralelismo dinâmico permite a criação de blocos de *threads* através de chamadas da própria GPU, permitindo que ela gere trabalho, controle o fluxo de trabalho e de escolonamento, sem ser necessário chamadas da CPU.

(a) (b)

Figura 4.6: A Figura (a) representa o novo SM da arquitetura Maxwell, a Figura (b) representa um diagrama da arquitetura Maxwell com 32 SMs [97].

4.4.4 Arquitetura Maxwell

A arquitetura Maxwell [97] foi apresentada pela NVidia em 2014 como a substituta da arquitetura Kepler (Seção 4.4.3) e, em 2016, foi substituída pela arquitetura Pascal (Seção 4.4.5).

A Figura 4.6a ilustra o SM da arquitetura Maxwell. Nele, o SM é particionado em 4 regiões de processamento com 32-CUDA core (suportam ponto flutuante de 32 bits), totalizando de 128 32-CUDA *cores* por SM. Na arquitetura CUDA, os SMs escalonam as *threads* em grupos de 32 *threads* chamadas *warp* [93]. Cada SM possui um hardware chamado de *Warp Scheduler* (WS). Na arquitetura Maxwell, cada partição de processamento do SM possui 1 WS, junto de um *buffer* de instruções, 8 unidades LD/ST e SFUs.

A Figura 4.6b representa um diagrama da arquitetura Maxwell com 16 SMs. Ela é divida em 4 *Graphics Processing Clusters* (GPCs). A porção azul clara central da figura representa o cache L2. As porções azul claras nos extremos da figura representam os o

cache L1, *warp schedulers* (WSs) e *dispatch units* (DU).

Existem algumas diferenças e semelhanças entre a arquitetura Maxwell e seu predecessor, a arquitetura Kepler (Seção 4.4.3). Podemos citar dentre as diferenças [97]: uma redução no número de *cores* por SM, porém cada core possui um desempenho $1,4\times$ maior, ocupando uma área menor. A arquitetura ainda é fabricada com transistores de 28 nm e possui um CUDA *Compute Capability* 5.0 ou 5.2 que também permitem o número máximo de *threads* por multiprocessador de 2048, mas permitem 32 blocos de *threads*.

4.4.5 Arquitetura Pascal

A arquitetura Pascal [99] foi apresentada pela NVidia em 2016, substituindo a arquitetura Maxwell (Seção 4.4.4). O processador GP100 desta arquitetura, possui 6 GPCs, cada um possui 5 SMs, totalizando 60 SMs no total. O SM possui 128 32-CUDA *cores*, 96 KB de memória compartilhada e um total de 48 KB de cache L1.

A Figura 4.7 ilustra o SM da arquitetura Pascal. O SM é dividido em 2 regiões. Ao centro, as 6 primeiras colunas ilustram 32 32-CUDA e 16 64-CUDA core. As duas últimas colunas são os 8 LD/ST e 8 SFUs. Ao topo da figura estão ilustrados o *cache* de instruções, o *buffer* de instruções além do WS e DU.

As placas da arquitetura Pascal são fabricadas utilizando tecnologia de 16 nm, permitindo um aumento no número de transistores. A arquitetura Pascal possui CUDA *Compute Capability* 6.0 a 6.2. A principal novidade desta versão são operações atômicas de ponto flutuante de 64-bit em memória global ou compartilhada.

Figura 4.7: SM da arquitetura Pascal [99].

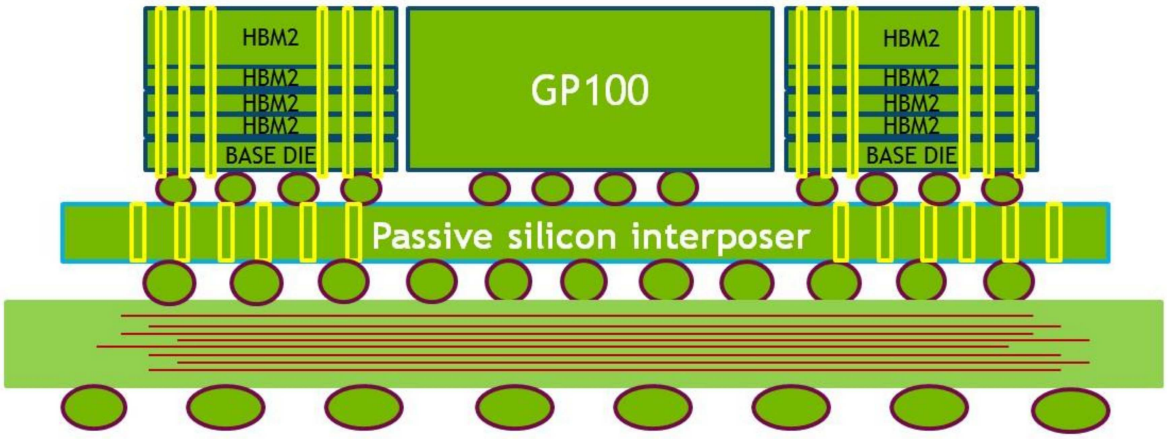


Figura 4.8: Esquema de conexão ilustrando as conexões HBM2 com o processador [99].

A Figura 4.9 representa um diagrama da arquitetura Pascal dividida em 6 GPCs, com um total de 60 SMs, 3.840 CUDA *cores* de precisão simples. A porção azul clara e central da figura representa o cache L2, cercado pelos SMs da arquitetura (Figura 4.7).

Dentre as diferenças existentes entre a arquitetura Pascal e sua predecessora, a arquitetura Maxwell (Seção 4.4.4), podemos destacar o significativo aumento na banda de memória utilizando High Bandwidth Memory 2 (HBM2).

A Figura 4.8 ilustra um esquema sobre as conexões do HBM2 e o processador GP100. Um circuito de memória tradicional GDDR5 apresenta vários chips de memória e conexões ao redor do processador da GPU. O HBM2 inclui uma ou mais *stacks* verticais de múltiplos *dies* de memória conectados por fios microscópicos. Então, um *passive silicon interposer* é utilizado para conectar a memória e o *die* do processador. Esse novo esquema de conectar a memória permite uma banda 3 vezes maior que as gerações anteriores. A GPU Tesla P100, com arquitetura Pascal, foi a primeira placa produzida comercialmente com HBM2. Porém algumas placas da arquitetura Pascal, como a Geforce GTX 1080, utilizam a memória GDDR5 tradicional [98].

4.4.6 Quadro Comparativo das Arquiteturas CUDA

Cada arquitetura CUDA possui um grande número de modelos de GPUs lançados, com diferenças de preço, memória e poder computacional. A Tabela 4.1 apresenta a comparação das arquiteturas CUDA. Para efeitos de comparação, utilizamos como referência as placas Tesla de cada arquitetura, que estão no patamar entre as de maior poder computacional dentre as placas de cada geração.

O número de SM em cada GPU aumentou 1,6 vezes entre a arquitetura Maxwell e Kepler, mas aumentou 2,33 vezes entre a Pascal e a Maxwell. Isso se deve principalmente

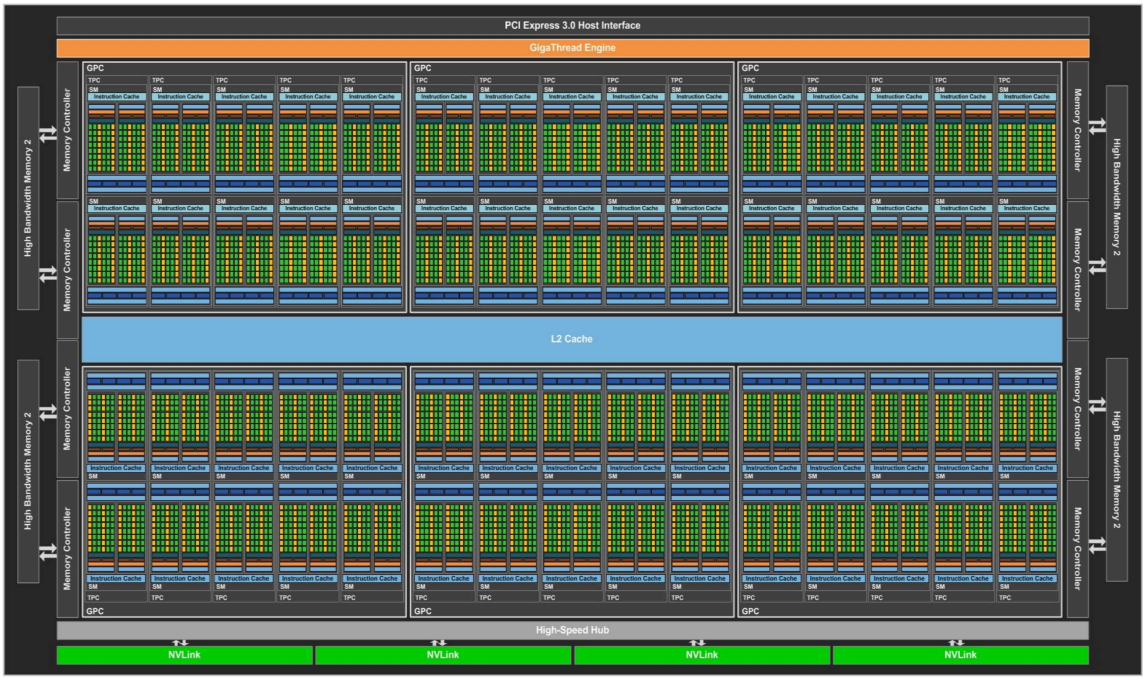


Figura 4.9: Diagrama da arquitetura Pascal com 60 SMs, do processador GP100 [99].

à mudança de tecnologia de produção, de 28 nm para 16 nm. Além disso, a nova interface de memória requer menos espaço, o que permite aumentar a área para os SMs [99].

Quando comparamos os CUDA core de precisão simples (PS), há uma tendência de queda no número de *cores* por SM, porém o número total por GPU tende a aumentar. Já para os CUDA core de precisão dupla (PD), a Maxwell apresentou uma queda no número de *cores*, enquanto a Pascal apresenta 1,86 vezes mais *cores* do que a Kepler.

Analisando o GFLOPs de cada uma das placas, as operações de precisão simples (GFLOPs PS) aumentam 1,35 vezes e 1,54 vezes, quando comparadas as arquiteturas Maxwell/Kepler e Pascal/Maxwell, respectivamente. Porém, para as operações de precisão dupla (GFLOPs PD), há uma queda entre a arquitetura Maxwell e Kepler.

Os *clocks* de cada geração de placa apresentam um crescimento. O *clock* base aumentou no total de 745 MHz para 1.328 MHz, e o *clock boost* aumentou de 810/875[1] MHz para 1.480 MHz. Em todos os testes de GFLOPs, os valores são medidos com a placa utilizando o *clock boost*. A interface de memória HBM2 (Seção 4.4.5) é um dos diferenciais da arquitetura Pascal em relação as anteriores que utilizam GDDR5. Porém a arquitetura Pascal apresenta uma queda na capacidade máxima de memória da placa, com uma redução de 24 GB para 16 GB. O tamanho total do cache L2 aumentou 2,0 e 1,33 vezes e o número de transistores apresentou um crescimento de 1,12 e 1,91 vezes comparando as arquiteturas Maxwell/Kepler e Pascal/Maxwell, respectivamente.

[1]Configurável por software, através da NVidia Management Library[94].

Tabela 4.1: Comparação entre as arquiteturas CUDA, utilizando como referências as placas Tesla [99].

Arquitetura	Kepler	Maxwell	Pascal
GPU	Tesla K40	Tesla M40	Tesla P100
Processador Gráfico	GK110	GM200	GP100
SMs	15	24	56
CUDA core por SM (PS)	192	128	64
CUDA core por GPU (PS)	2.880	3.072	3.584
CUDA core por SM (PD)	64	4	32
CUDA core por GPU (PD)	960	96	1.792
Clock Base	745 MHz	948 MHz	1.328 MHz
Clock Boost	810/875 MHz	1.114 MHz	1.480 MHz
GFLOPs (PS)	5.040	6.840	10.600
GFLOPs (PD)	1.680	210	5.300
Interface de Memória	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memória Máxima	12 GB	24 GB	16 GB
Cache L2	1.536 KB	3.072 KB	4.096 KB
Número de Transistores	7,1 bilhões	8,0 bilhões	15,3 bilhões
Tamanho dos Transistores	28 nm	28 nm	16 nm
Compute Capability	3.5	5.2	6.0

Capítulo 5

Alinhamento Múltiplo Primário em Arquiteturas Paralelas

Nesse capítulo, serão discutidas propostas para execução de ferramentas paralelas de Alinhamento Múltiplo de Sequências. Na Seção 5.1 serão apresentadas propostas para o alinhamento múltiplo heurístico em plataformas de alto desempenho e na Seção 5.2 serão apresentadas propostas para o alinhamento múltiplo exato em algumas dessas plataformas. Na Seção 5.3 serão apresentados e discutidos quadros comparativos das propostas. Na Seção 5.4 serão apresentadas estratégias paralelas que utilizam o algoritmo A-Star para o alinhamento múltiplo de sequências. Finalmente, na Seção 5.5, quadros comparativos das estratégias que utilizam A-Star em outros domínios são discutidos.

5.1 Propostas de Alinhamento Heurístico

5.1.1 MT-ClustalW

K. Chaichoompu *et al.* [10] criaram em 2006 uma versão do ClustalW (Seção 2.3.3.1) totalmente *multithreaded* chamada MT-ClustalW. Ela é baseada em outra implementação paralela chamada ClustalW-SMP, criada por O. Duzlevski [20].

A otimização proposta pelo MT-ClustalW foi realizada na fase 2, pois os autores detectaram que era a fase que mais demandava tempo. O código e fluxograma de execução propostos são apresentados na Figura 5.1. Nele, os dois *loops* internos foram movidos para *threads*. São criadas *thread_num threads* que recebem um sinal da *thread* principal e aguardam a execução. Quando todas as *threads* estão ocupadas, a *thread* principal aguarda que uma delas termine a execução.

O código foi implementando em C e utilizou a biblioteca *pthread* para obter paralelismo. Os testes foram realizados em uma estação com processador Intel Pentium D

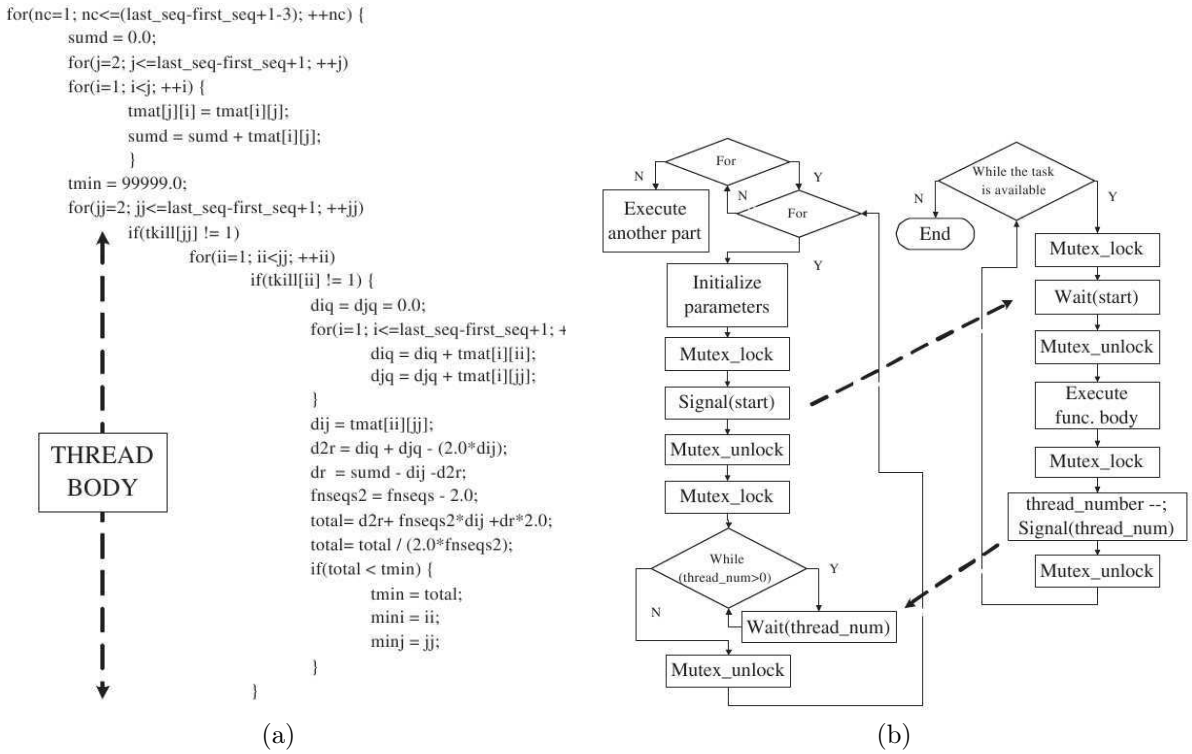


Figura 5.1: Código *multithread* do MT-ClustalW e fluxograma[10].

Dual-Core com 2,8 GHz e 2 GB de memória RAM. O sistema operacional utilizado foi o MS Windows XP PRO SP 2, e a máquina não possuía nenhuma outra aplicação em execução.

Os tempos de execução da fase 2 são comparados entre o MT-ClustalW e o ClustalW-SMP. Em todos os casos, o MT-ClustalW apresentou desempenho melhor do que o do ClustalW-SMP. Na comparação dos métodos, o *speedup* do MT-ClustalW em relação ao Clustal-SMP é maior para sequências curtas, de até 200 aminoácidos, atingindo um *speedup* máximo de $1,65\times$. Isto acontece porque a fase paralelizada do algoritmo MT-ClustalW possui tempo de execução proporcional ao número de sequências. As outras fases são proporcionais ao número de sequências e seu comprimento. Assim, quando o comprimento das sequências aumenta, mais tempo é gasto em fases que são iguais em ambos algoritmos, desta forma o *speedup* diminui.

5.1.2 ClustalW-MPI

Em 2003, K. Li publicou uma estratégia para execução em *clusters* (Seção 4.1.4) chamado ClustalW-MPI [73]. No ClustalW-MPI a primeira e a terceira fases do ClustalW são paralelizadas.

A primeira etapa do algoritmo ClustalW é fácil para paralelização, pois possui uma granularidade grossa e as comparações par-a-par são independentes entre si. Já as duas

últimas etapas são um desafio maior para paralelizar pois possuem uma razoável dependência entre os dados.

Para a primeira fase, é utilizada uma estratégia de escalonamento chamada *fixed-size chunking* (FSC), onde lotes de tarefas de um tamanho fixo são destinados para os processadores disponíveis.

Com os alinhamentos par-a-par calculados, uma árvore guia é produzida para servir de topologia para o alinhamento final (Seção 2.3.3.1). Algumas modificações foram realizadas no algoritmo para que ele pudesse ser executado em $O(N^2)$, obtendo o mesmo resultado do algoritmo original. Segundo os autores, essa fase não foi paralelizada pois normalmente utiliza menos de 1% do tempo total de execução.

Na última fase, os nós externos da árvore são alinhados paralelamente. Esta abordagem obviamente depende da topologia da árvore obtida. No melhor caso, é possível obter um *speedup* de $N/logN$.

Nos testes, foi utilizado um *cluster* composto por oito processadores *dual-core*, Pentium III, 800 MHz conectados por Fast Ethernet. Foram alinhadas 500 sequências de proteínas de um tamanho médio de 1.100 aminoácidos. Neste teste, cada processador recebe um lote de 80 alinhamentos em pares para realizar.

No artigo, são mostrados os tempos de execução e os *speedups* obtidos com o ClustalW-MPI para 500 sequências. O *speedup* do cálculo das distâncias é quase linear sendo 15,8x usando 16 processadores. Já para a fase 3 (alinhamento progressivo), é obtido um *speedup* de 4,3x utilizando 16 processadores.

5.1.3 ClustalW em FPGA

T. Oliver *et al.* [100] criaram em 2006 uma abordagem paralela baseada em FPGA (Seção 4.2) para a execução do ClustalW (Seção 2.3.3.1).

Analisando as três fases do algoritmo ClustalW, os autores concluíram que mais de 90% do tempo total de execução era gasto na primeira fase (alinhamentos em pares) e esta foi a fase adaptada para *hardware* reconfigurável.

O alinhamento em pares que utiliza programação dinâmica e exibe uma alta regularidade. Sendo assim, pode ser eficientemente mapeado em um *array* de pequenos elementos de processamento (PE). Cada caractere de uma sequência (sequência de busca) é atribuído a um PE e os caracteres da segunda sequência (sequência *database*) percorrem os PEs. Desta forma, cada PE potencialmente calcula uma célula da matriz de programação dinâmica a cada ciclo de clock. Foi necessário particionar a sequência de busca, pois o comprimento de uma típica sequência de aminoácido é maior que a quantidade de PEs que a placa FPGA utilizada suporta.

Apesar da eficiência no cálculo das matrizes, é necessário obter o alinhamento das sequências, e a operação de *traceback* não pode ser tão eficientemente paralelizada em FPGA. Por isso, essa fase foi executada em CPU.

Nos testes, foi utilizada uma placa Xilinx Virtex II XC2V600, onde foi possível acomodar 92 PEs a um clock de 34 MHz. A implementação conseguiu um desempenho de aproximadamente 1 GCUPS (Bilhões de Atualizações de Células da Matriz de Programação Dinâmica por Segundo). No artigo, são comparados os resultados do desempenho do ClustalW em FPGA e a solução em *software* do ClustalW, executando em um Pentium IV de 3 GHz e 1 GB de memória RAM.

Os resultados mostram que, para a primeira fase, foi possível obter um *speedup* de 45× a 50×, quando comparado com a execução sequencial.

Da mesma maneira que no ClustalW-MT, os melhores desempenhos são obtidos com sequências mais curtas. Observou-se também que o tempo de execução total foi acelerado em mais de dez vezes, enquanto o *speedup* da fase 1 foi ao menos 45x. Sendo assim, os autores concluíram que o tempo de execução da fase 1 (alinhamentos em pares) não é o que mais influencia no resultado final e a otimização da terceira fase é citada como trabalhos futuros.

5.1.4 GPU-ClustalW

GPU-ClustalW [75] é um exemplo de algoritmo que executa em GPUs que não possuem suporte à computação de propósito geral, ou seja, o ClustalW (Seção 2.3.3.1) foi re-escrito em 2006 de forma a simular uma renderização de vídeo. Uma típica renderização segue uma ordem fixa de três estágios de processamento, chamada de *pipeline* gráfico.

O primeiro estágio, de processamento de vértices, transforma um conjunto de vértices tridimensionais em uma tela de vértices de duas dimensões. Na segunda fase, o rasterizador converte a representação geométrica dos vértices em coordenadas de uma tela. No final, o processador de fragmentos constrói uma cor para cada pixel, lendo os pixels de uma memória de textura. As GPUs permitem programar o processador de vértices e de fragmentos. Porém, os programas de fragmentos podem ser utilizados para implementar qualquer operação matemática em um ou mais vetores de entrada (texturas ou fragmentos) para computar a "cor" de um pixel.

Para calcular o alinhamento múltiplo, apenas a primeira fase do algoritmo ClustalW foi paralelizada, pois os autores concluíram que mais de 93% do tempo gasto no algoritmo era na primeira fase, conforme alguns testes de 200 a 1.000 sequências.

Na implementação, é aproveitado o fato de que todos os membros de uma antidiagonal da matriz podem ser calculados em paralelo. Como a célula de posição (i, j) possui como dependência os vizinhos $(i-1, j)$ $(i, j-1)$ e $(i-1, j-1)$, sempre são mantidas em

memória três antidiagonais em três *buffers* diferentes e é utilizado um método cíclico para sobrescrevê-los a cada iteração. As diagonais $k-1$ e $k-2$ são armazenadas como texturas de entrada e a diagonal k é o alvo a ser renderizado. Na iteração seguinte, k se torna $k-1$, $k-1$ se torna $k-2$ e $k-2$ se transforma em k.

Considerando um conjunto de n sequências, elas são ordenadas de acordo com o tamanho, de modo a calcular todas as $n \times (n-1)/2$ comparações em pares. Para melhor desempenho, as sequências são armazenadas em texturas 2D e múltiplas comparações em pares podem ser feitas ao mesmo tempo.

No algoritmo, os valores $H(i,j), H(i-1,j), H(i,j-1)$ são calculados para cada célula e armazenados em um canal-A de um pixel de cor RGBA em dois alvos de renderização separados. Cada pixel armazena uma conta matemática diferente para calcular o valor da célula usando Smith-Waterman (Seção 2.2.2) e considerando penalidades *affine-gap* [87].

O algoritmo proposto foi implementado com a linguagem de programação GLSL (*OpenGL Shading Language*) e foi testado com a placa de vídeo NVidia GeForce 7800 GTX, com 627 MHz, 512 MB de RAM a uma velocidade de 1,83 GHz, 8 processadores de vértices e 24 processadores de fragmentos. A estação onde os testes foram realizados possuía processador Intel Pentium4 3,0 GHz, 1 GB RAM, executando Windows XP. Nos testes, foi possível obter um *speedup* de 10×, comparado a uma solução em *software*.

Outras ferramentas de alinhamento múltiplo, como T-Coffee ([90]) e MUSCLE [21], também possuem fases que executam alinhamentos em pares e, de uma forma semelhante à apresentada no artigo, é proposto como trabalhos futuros a otimização dessas ferramentas.

5.1.5 CUDA ClustalW

O CUDA ClustalW [52], publicado em 2015, propõe o uso de paralelismo *intra-task* para calcular o alinhamento múltiplo de sequências de proteína com ClustalW (Seção 2.3.3.1) em uma GPU ou em duas GPUs em uma mesma máquina. A fase 1 do ClustalW (alinhamento em pares) é paralelizada. As sequências de entrada são classificadas pelo tamanho e os pares de sequências são atribuídos de maneira uniforme às GPUs e aos blocos de *threads* de cada GPU. Na GPU, os pares de sequências a serem comparados são mantidos em memória compartilhada (*shared*). É executado um algoritmo adaptado de Hirschberg [47] e cada alinhamento em par é atribuído a um bloco de *threads*, explorando o paralelismo da frente de onda e, portanto, calculando cada anti-diagonal em paralelo.

O CUDA ClustalW foi implementado em CUDA, C, e OpenMP (versão duas GPUs) e testado em uma máquina com Intel Xeon X5550, 24 GB RAM, sistema operacional CentOS v.5.3 e duas GPUs NVidia Tesla M2050. Foram comparadas sequências de até 1.523 aminoácidos, com 64 *threads* por bloco. Os resultados obtidos com o CUDA ClustalW foram comparados com o ClustalW em GPU, conseguindo-se um *speedup* de até $32,84\times$,

quando comparadas 1.000 sequências de 1.523 aminoácidos. Na comparação de 1.000 de menor tamanho (97), o speedup cai para $4,47\times$.

5.1.6 MSA-CUDA

O MSA-CUDA [76] é um algoritmo proposto em 2009, que utiliza GPU para executar o ClustalW (Seção 2.3.3.1). Ele é construído utilizando CUDA (Seção 4.4.1). Nesta implementação, todas as três fases do algoritmo ClustalW foram paralelizadas.

Na primeira fase do algoritmo, utiliza-se Smith-Waterman (Seção 2.2.2) para se fazer a comparação de todos os pares de sequências. Nesta fase, a computação das distâncias dos pares de sequências é definida como uma tarefa. Foram testadas duas abordagens: paralelização *inter-task* e paralelização *intra-task* [76], para paralelizar todas as fases do algoritmo.

Na paralelização *inter-task*, cada tarefa é atribuída para exatamente uma *thread* e *dimBlocks* tarefas são executadas em paralelo, dentro do bloco de *threads*. Na paralelização *intra-task*, cada tarefa é atribuída para um bloco de *threads*, que contribuem para finalizar a tarefa em paralelo, utilizando as propriedades das anti-diagonais da matriz. A otimização *inter-task* exige muitas tarefas para ser eficiente. Caso existam poucas tarefas (≤ 100), os autores acreditam que a paralelização *intra-task* é preferível.

Várias estratégias foram utilizadas para se atingir um bom resultado. Primeiramente, ordena-se todas as sequências pelo tamanho. Para se obter um melhor uso da largura de banda, todas as *threads* utilizam um acesso padronizado (*coalesced*) à memória [96]. Ambos métodos de paralelização utilizam memória constante para armazenar parâmetros de somente leitura, assim como a matriz de substituição (Seção 2.1.1). A matriz de substituição é carregada para a memória compartilhada ao ser acessada por uma *thread*, pois diferentes *threads* frequentemente acessam o mesmo dado da matriz de substituição. A memória de textura é utilizada para armazenar as sequências ordenadas. Estas são as estratégias apresentadas para a primeira fase do algoritmo.

Na segunda fase, a árvore guia é construída através do método *neighbor-joining* (NJ). Esta fase pode ser dividida em duas sub-fases: reconstrução de uma árvore sem raiz e criação da raiz da árvore NJ com o cálculo do peso das sequências. Um bloco de *threads* é responsável por calcular as diferenças entre os filhos de cada nó da árvore e cada *thread* é responsável por calcular em um subconjunto separado de nós. Nesta etapa, a memória compartilhada é utilizada para armazenar resultados e a memória de textura é utilizada para armazenar a estrutura da árvore.

Para fazer a paralelização da terceira fase, cada possível alinhamento possui uma *flag* indicando se já foi ou não alinhado. Caso um alinhamento possua dois filhos que já foram alinhados, ele é marcado para ser alinhado na próxima iteração. Caso contrário, ele deverá

aguardar até que os dois filhos sejam alinhados. O processo é repetido até que todos os nós tenham sido alinhados.

Três algoritmos foram utilizados nos testes: ClustalW Sequencial, MSA-CUDA e ClustalW-MPI (Seção 5.1.2) executando-se em um *cluster* de 32 processadores.

O MSA-CUDA foi implementado em CUDA C e testado em uma estação com uma placa NVidia GeForce GTX 280 e 1 GB RAM e um processador AMD Opteron 248 2,2 GHz.

Nos resultados obtidos, a fase 1 do algoritmo claramente possuiu um melhor desempenho utilizando a abordagem *inter-task*. Por isso, se existir memória suficiente na GPU, o MSA-CUDA escolhe a otimização *inter-task* para esta fase. Os *speedups* são maiores em um número menor de sequências, mas de comprimento maior, pois neste caso o esforço computacional é maior.

Na fase dois, era esperado que o tamanho das sequências não influenciasse no desempenho e por isso os melhores *speedups* foram obtidos nessa fase com o aumento do número das sequências. Na terceira fase, existe uma grande divergência entre os *speedups* obtidos. Isto acontece pois a topologia da árvore guia e o tamanho das sequências interferem no desempenho. Normalmente, um número grande de sequências longas significa um melhor *speedup*.

Finalmente, os autores fazem a comparação entre o MSA-CUDA, o ClustalW-MPI e o ClustalW sequencial. Para um grande número de sequências, o desempenho do ClustalW-MPI cai muito pois a segunda fase exige grande esforço computacional e o ClustalW-MPI não paraleliza esta fase. Entretanto, mesmo no caso em que o ClustalW-MPI apresenta um bom desempenho em relação ao sequencial, uma única GPU utilizando MSA-CUDA possui um *speedup* $41,53\times$. Este desempenho é superior ao ClustalW-MPI em um *cluster* com 32 processadores, que possui um *speedup* de $39,24\times$. Isso mostra que esta abordagem em GPU possui uma relação de custo/desempenho muito boa.

5.1.7 G-MSA

G-MSA [4] é uma proposta de implementação do algoritmo T-Coffee (Seção 2.3.3.1) em GPU. Inicialmente, as sequências de entrada são recebidas pela CPU e a GPU é usada para gerar os alinhamentos globais de todos os possíveis pares de sequências, bem como os 10 melhores alinhamentos locais sem sobreposição de todos os possíveis pares de sequências. A seguir, a fase de construção da biblioteca primária (PL) também é executada em GPU. Devido à grande necessidade de memória, essa fase é dividida em subproblemas, chamadas de janelas. Cada janela é processada por vez em GPU. A seguir, a biblioteca estendida (ES) também é gerada na GPU, usando uma estratégia que otimiza o uso de memória.

O algoritmo proposto foi implementado em C e CUDA e testado em 2 placas NVidia GTX-480. Os resultados foram obtidos com 25 a 500 sequências, cujo o tamanho variava de 100 a 420 aminoácidos. O G-MSA foi comparado com versões sequenciais do T-Coffee, Clustal-W e ClustalW-MPI (Seção 5.1.2), entre outros. Quando comparado ao T-Coffee sequencial, o G-MSA obteve um *speedup* de 193x, usando uma única GPU e fazendo o alinhamento múltiplo com a biblioteca estendida. O G-MSA, usando somente a biblioteca primária, conseguiu ser 10x mais rápido que o ClustalW sequencial.

5.1.8 CMSA

O CMSA [11] foi proposto em 2017 e visa alinhar sequências longas ou curtas de DNA ou RNA em plataformas heterogêneas contendo CPU e GPU. Considera que todas as sequências a serem comparadas possuem alta similaridade e vêm de um único ancestral comum. Para esse tipo de problema, é usado um tipo especial de árvore, chamada "*star*" [2], que conecta todas as sequências diretamente à raiz e o método é chamado "*center star*".

A computação de alinhamentos múltiplos com "*center star*" é feita em três etapas: (a) achar a sequência de centro; (b) alinhamento par a par da sequência de centro com as demais; (c) tratamento de *gaps*. No CSMA, a fase (a) é executada em CPU e processa utilizando uma representação binária das sequências (dois bits para cada nucleotídeo), que são particionadas em segmentos disjuntos de 8 caracteres (16 bits). É feita uma contagem de ocorrência de cada segmento nas sequências a serem comparadas e o escore de cada sequência é a soma da contagem de ocorrência de cada um de seus segmentos. A sequência de centro é a sequência cujo escore é máximo.

A fase (b) é executada tanto em CPU como GPU, utilizando paralelismo a grão grosso, ou seja, cada *thread* compara uma sequência diferente com a sequência de centro, usando programação dinâmica. Para determinar o poder computacional da CPU e GPU, é feita uma pré-computação com poucas sequências de tamanho pequeno e obtido o poder computacional relativo. A fase (c) é bastante rápida e é executada unicamente em CPU.

O CMSA foi implementado em C, CUDA e OpenMP e testado em uma máquina contendo Intel Xeon E5-2620 2,4 GHz, 32 GB RAM, com o sistema operacional CentOS 6.5 e uma GPU NVidia Tesla K40, com 2.880 CUDA *cores*. Foram usados 5 conjuntos de sequências, indo de 672 sequências de tamanho médio 16.569 a 500.000 sequências de tamanho médio 748. O CMSA foi comparado com o Halign [142], que também usa o método *center star*, porém se executa totalmente em CPU. *Speedups* de até $16,00\times$ foram obtidos. Na comparação do CSMA somente GPU com GPU/CPU, os autores mostram que em alguns casos a comparação com GPU/CPU é mais lenta. Eles concluem que essa

estratégia deve ser utilizada somente quando há um grande volume de dados (e.g., 500.000 sequências de tamanho médio 748).

5.2 Propostas de Alinhamento Exato

5.2.1 MSA exato com MPI

Helal *et al.* [42–44] apresentam em 2009 estratégias em MPI para calcular em um *cluster* de computadores o alinhamento múltiplo exato, reduzindo o espaço de busca sem usar o Carrillo-Lipman (Seção 2.3.1.1) ou A-Star (Seção 2.3.2). Em [44], Helal *et al.* utilizam uma abordagem mestre/escravo, onde o espaço é dividido em ondas de computação que contém partições equidistantes de um mesmo ponto de origem. Essas ondas são pré-processadas para acelerar a computação. O processo mestre é responsável por consumir recursos e escalonar as partições nos processadores. Há um custo de comunicação dentro e através das ondas de processamento. Na abordagem distribuída [42, 43], não existe um processador mestre e o escalonamento de partições se dá através da interação entre processadores vizinhos.

O algoritmo proposto em [44] foi implementado em C e MPI, podendo ser executado em ambientes com até 64 processadores. Foi utilizado um SunFire X2200 com 2 processadores AMD Opteron *quad core* de 2,3 GHz, 512 Kb L2 cache e 2 MB L3 em cada processador. A estação estava equipada com 8 GB RAM e 8 *cores* foram utilizados.

Nos resultados experimentais, o número de sequências comparadas é muito pequeno (até 5) e seu tamanho máximo também é pequeno (até 41). Como o algoritmo implementado é exato e, portanto, executa-se em tempo exponencial, pode ser notada a grande diferença entre os tempos de comparação de três sequências e os tempos de comparação de cinco sequências.

5.2.2 MSA exato em FPGA

Utilizando FPGA (Seção 4.2), Yamaguchi *et al.* [79] implementaram o algoritmo Carrillo-Lipman (Seção 2.3.1.1) para o alinhamento múltiplo exato. Sua solução consiste em utilizar o FPGA para calcular o escore visitando todos os vizinhos de uma célula do espaço de busca e utilizar dados calculados em *software* para auxiliar na redução do espaço de busca.

No circuito implementado, a programação dinâmica N-dimensional é realizada repetindo-a em duas dimensões e variando as outras dimensões. Considere X, Y, Z os tamanhos de três sequências nos eixos x, y, z respectivamente. Sejam W_x, W_y, W_z as partes de uma sequência que podem ser processadas continuamente, sem ser necessária

qualquer entrada ou saída de dados da placa. A Figura 5.2 mostra como a programação dinâmica em três dimensões é realizada através de repetições de uma programação dinâmica de duas dimensões. A área em processamento é chamada de janela de varredura [79].

Para implementar o método de Carrillo-Lipman, seja P, assim como mostrado na Seção 2.3.1.1, o conjunto do espaço de busca reduzido. O limite do espaço de busca que deve ser calculado para obter o alinhamento é calculado em CPU e carregado em bancos externos de memória na placa FPGA. Antes de iniciar o cálculo de alguma janela de varredura, o algoritmo decide se a janela de varredura atual deve ou não ser calculada. Desta forma, o espaço de busca é reduzido.

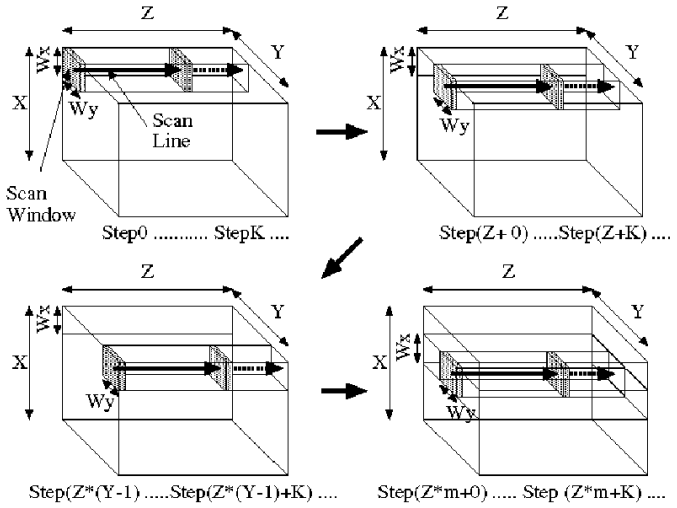


Figura 5.2: Programação dinâmica em três dimensões [79].

Dois circuitos foram implementados, um para alinhamento de quatro sequências e outro para alinhamento de cinco sequências com frequências de 36 MHz e 31 MHz, respectivamente. A frequência baixa de operação é causada pelos seletores para escolher entre $2^n - 1$ candidatos.

O MSA em FPGA foi comparado com o MSA 2.0 (Seção [31]), executado em um processador Intel Pentium 4 3 GHz com 2 GB de memória. O *speedup* obtido é influenciado pela similaridade das sequências comparadas. Quando as sequências são muito semelhantes, a solução em *software* funciona muito bem, pois grande parte do espaço de busca não é calculado. Por outro lado, grandes blocos de processamento são sempre executados em FPGA, resultando em um desempenho inferior neste caso.

Por não ser sempre melhor, é aconselhável utilizar a solução em FPGA em paralelo com a solução em *software*, pois algumas execuções podem ser feitas em menor tempo em CPU. Para problemas de baixa similaridade foi possível obter *speedups* de até 50× quando comparada à solução em *software*.

5.2.3 MSA exato com Carrillo-Lipman (MSA-GPU)

Em um trabalho anterior, nós propusemos o MSA-GPU [124], uma solução que utiliza GPU (Seção 4.4) para resolver o alinhamento múltiplo exato. Para reduzir o espaço de busca foi utilizado o método de Carrillo-Lipman (Seção 2.3.1.1). Foram propostas duas estratégias: *fine-grained* e *coarse-grained*.

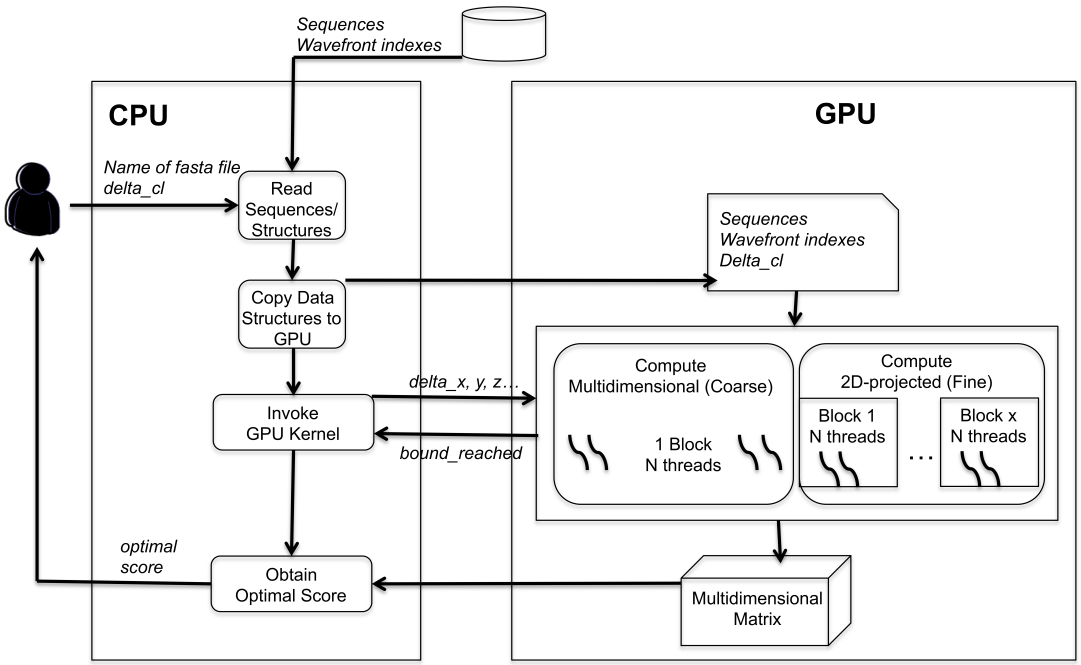


Figura 5.3: Visão geral das estratégias [124].

A Figura 5.3 fornece uma visão geral da solução. Primeiramente, o usuário provê as sequências e o limite de Carrillo-Lipman δ_{msa} (*delta_cl*) (Seção 2.3.1.1). Então, é feita a leitura das sequências, a inicialização e cópia dos dados para a GPU. Após isso, os *kernels* da GPU são executados de acordo com a estratégia utilizada. Após a execução de cada *kernel*, a variável *bound_reached* é utilizada para informar se o limite Carrillo-Lipman foi atingido e as células podem ser descartadas do espaço de busca. Ao final de toda a execução, o escore ótimo é retornado ao usuário.

Na estratégia *Coarse-Grained* busca-se aumentar o paralelismo calculando o maior número de células possível do espaço de busca simultaneamente. Porém, devido a limitações de recursos do *hardware*, nem sempre é possível criar um *wavefront* multidimensional do tamanho do espaço de busca, e neste momento inicia-se o percorrimento do espaço de busca projetando-se em uma dimensão. Foi observada uma queda de paralelismo nesta estratégia. Por este motivo, decidiu-se criar uma outra estratégia, de granularidade mais fina, e que percorre o espaço de busca em projeções de diagonais 2D conforme a proposta de Yamaguchi et al. (Seção 5.2.2).

Nesta estratégia de granularidade fina, o espaço de busca é otimizado para ser percorrido conforme projeções de diagonais 2D, de forma que quando todo o espaço de busca é percorrido em uma diagonal, ocorre o deslocamento em um sentido e então o processo é reiniciado. A granularidade deste método é maior pois o cálculo de cada célula não é feita apenas por uma *thread*, mas por $2^n - 1$, onde n é o número de sequências. Desta forma, cada *thread* é responsável por uma célula na equação de recorrência, e ao final elas fazem uma comparação em paralelo para decidir qual valor é o melhor para ser utilizado na célula.

O MSA-GPU foi implementado em CUDA C e testado em uma estação com uma placa NVIDIA GeForce GTX 280 e 1 GB RAM e um processador AMD Opteron 248 2,2 GHz. Os resultados foram coletados com conjuntos de 3 sequências reais e sintéticas de aminoácidos e mostraram que para as sequências de menor tamanho, a estratégia *fine-grained* consegue melhores resultados que a estratégia *coarse-grained* mesmo quando emprega menos *threads*. Quando o tamanho das sequências é aumentado, a estratégia *coarse-grained* ultrapassa a *fine-grained* com um bloco, pois o número de *threads* não é o suficiente para lidar com o paralelismo. Porém, para todas as sequências comparadas, a estratégia *fine-grained* com múltiplos blocos consegue os melhores tempos de execução entre as estratégias, com uma grande vantagem sobre as outras estratégias, conseguindo reduzir o tempo de 8 minutos e 55 segundos (*coarse-grained*) para 6,62 segundos (*fine-grained* multibloco).

A estratégia *fine-grained* com múltiplos blocos foi comparada com o programa MSA 2.0 (Seção 2.3.1.2). Os resultados mostram que, para instâncias pequenas do problema a execução do programa em CPU é muito rápida, obtendo melhores resultados do que a GPU. Porém, para os casos com menor similaridade, o programa MSA 2.0 não se executa eficientemente com os parâmetros padrão, e então foi necessária uma nova execução, com outros parâmetros. Para estes casos, o programa MSA 2.0 foi mais lento que a implementação em GPU, sendo que a GPU apresenta um *speedup* de $7,8\times$ para o caso de sequências de média similaridade e $6,02\times$ e $8,60\times$ para os casos de sequências de baixa similaridade, de comprimento médio e longo, respectivamente.

5.2.4 MSA exato com A-Star em *Cluster*

Niewiadomski *et al.* [89], apresentaram duas estratégias, uma serial e uma paralela para resolver o problema do alinhamento múltiplo exato de sequências. Nele, é usada uma variação do A-Star (Seção 2.3.2) chamada Frontier A* (Seção 2.3.2), combinada com a otimização DDD (Seção 2.3.2).

A principal contribuição do trabalho é o PFA*-DDD, uma versão paralela do algoritmo que combina o Frontier A* e o DDD, executando-se em um *cluster* de computadores. Uma

dificuldade em criar a versão paralela é em como distribuir o trabalho entre as estações de maneira eficiente. A ideia da estratégia proposta é particionar a fila de tarefas em grupos disjuntos entre as n estações do *cluster*. Todo o trabalho envolvendo nós do i-ésimo grupo é processado pela i-ésima estação de *cluster*. Cada nó é representado por um inteiro e o particionamento inicial é feito dividindo-se o intervalo inteiro em n intervalos. A cada iteração o PFA*-DDD pode escolher entre manter o particionamento decidido, ou caso um dos nós acredite estar sobrecarregado, ele pode iniciar um procedimento para dividir novos intervalos entre o *cluster*.

O artigo mostra os resultados para a solução de duas instâncias do BAliBASE [130] em um *cluster* de 32 *cores*. Essas instâncias estão entre as mais difíceis da referência 1 do BAliBASE , apresentando *speedups* de até 32x usando 32 *cores*. No entanto, a implementação limita o número de sequências entre 3 e 8, de forma que limita os testes a serem realizados em instâncias maiores do problema.

5.2.5 MSA exato com A-Star e Memória Externa

Um dos principais fatores limitantes do algoritmo A-Star é a memória para manter a lista de nós abertos e fechados. Hatem *et al.* [37] propuseram uma estratégia com o uso de uma memória externa (disco) para poder solucionar o problema em instâncias mais difíceis.

A principal contribuição deste trabalho é propor uma estratégia que combina as expansões parciais do PEA* com a técnica de *hash* do HBDDD. Esse nova estratégia é chamado de Parallel External Partial Expansion A* (PE2A*) e foi testada em uma estação com 12 discos em paralelo, visto que o uso intensivo de disco do algoritmo visa ganho com I/O paralelo. São utilizadas 24 *threads* durante a execução deste teste.

Nos resultados experimentais, foram utilizadas todos os conjuntos de sequência da referência 1 do BAliBASE [130]. A estação utilizada estava equipada com dois processadores Intel Xeon X5660, 2,80 GHz e 6 *cores*, além de ter 12 GB de RAM e 12 discos de 320 GB.

Os resultados mostram que o PE2A* consegue reduzir em cerca de 60% o tempo de execução quando comparado a outra abordagem, o PA*-DDD [68]. A instância que mais tempo exigiu foi a 1pamA, resolvida pelo PA*-DDD em 32 dias e 14 horas, porém o PE2A* resolve em 9 dias e 8 horas. Os autores afirmam que foi a primeira vez que estas instâncias foram resolvidas utilizando *affine gap* em uma única máquina.

5.3 Quadro Comparativo de Estratégias MSA

Nesta seção são comparadas as soluções para o alinhamento múltiplo de sequências descritas nas Seções 5.1 e 5.2, Nessa comparação são considerados o número de sequências, o tempo obtido (se disponível), o ano do trabalho e o *speedup* (quando existir).

Tabela 5.1: Comparação entre as estratégias paralelas heurísticas em plataformas de alto desempenho

Ref.	Ano	Algoritmo	Arquitetura	Seqs. (Tam médio)	Tempo (s)	*Speedup*
[73]	2003	ClustalW-MPI	Cluster (16 *cores*)	500 (1.000)	13.000,0	9
[10]	2006	MT-ClustalW	SMP (8*threads*)	200 (200) 800 (800) 500 (500)	22,0 7.550,0 2.194,5	1,18 1,13 1,13
[100]	2005	ClustalW em FPGA	FPGA	200 (412) 1.000 (446) 600 (448,4)	14,7 399,5 181,2	13,30 11,80 12,08
[75]	2006	GPU ClustalW	GPU	1.000 (446) 600 (436,4)	680,7 311,9	6,90 6,92
[76]	2009	MSA-CUDA (ClustalW)	GPU	400 (856) 8.000 (73) 3.233 (392,8)	N/A	32,29 10,38 22,30
[4]	2012	G-MSA (T-Coffee)	GPU	25 (100) 500 (420)	5 (PL) e 5 (ES) 9 (PL) e 700 (ES)	10,0 193,0
[52]	2015	CUDA ClustalW	GPU	1.000 (97) 1.000 (1.523)	29,3 892,6	4,47 32,84
[11]	2017	CMSA (Center-Star)	CPU-GPU	672 (16.569) 500.000 (748)	44,7 23,1	3,19 16,00

A Tabela 5.1 compara as soluções de estratégias heurísticas em plataformas de alto desempenho. Como pode ser visto, o ClustalW foi o algoritmo mais adaptado para tais plataformas. Quanto ao tipo de plataforma usada, observamos que as soluções mais recentes são feitas em GPU e que, para o ClustalW, as soluções mais recentes conseguem comparar até 8.000 sequências em GPU ou até 500.000 sequências em CPU e GPU. Os melhores *speedups* são também obtidos em GPUs, sendo que alguns são comparáveis a um *cluster* de 32 *cores*.

Ainda na Tabela 5.1, uma solução que utiliza GPU [75] teve um *speedup* inferior quando comparado ao uso de *hardware* reconfigurável [100], mas alguns anos depois, com a evolução das GPUs e uma nova solução, *speedup* médio obtido foi maior em 2009 [76] e 2015 [52]. Também é possível notar a diferença entre o número de sequências permitida pelos métodos, chegando à 500.000 sequências em [11]. Os *speedups* citados são os apresentados nos artigos e servem apenas como uma noção da aceleração, não devendo ser feita a comparação direta entre eles. Os *speedups* reportados variam de acordo com a forma utilizada para medir tempo, com as sequências utilizadas, com o tamanho delas e a quantidade de sequências. Em cada trabalho, a forma de medir o tempo muitas vezes não era citada e os conjuntos de sequências utilizados eram diferentes.

A Tabela 5.2 apresenta as estratégias para execução de algoritmos exatos de alinhamento múltiplo de sequências em plataformas de alto desempenho. Como pode ser visto, as propostas paralelas para a execução do algoritmo exato comparam poucas sequências (até 10) e possuem um tempo de execução expressivo, podendo chegar a mais de 9 dias [37].

As arquiteturas de *hardware* reconfigurável apresentam um *speedup* considerável, tanto que a melhor ferramenta conhecida para o alinhamento ótimo utiliza FPGA [79]. Porém essa arquitetura possui uma perda grande de desempenho quando as sequências são similares, sendo consideravelmente mais lentas que a execução em *software* em uma única *thread*, com um *speedup* de $0,07\times$.

Quando utiliza-se o A-Star [89], pode-se observar que a execução em *cluster* possui um paralelismo que pode ser explorado, atingindo bons *speedups* com até 32 *cores*. Uma outra alternativa a essa solução são estratégias baseadas em memória externa [37], onde os nós são mantidos em memória e em disco.

Tabela 5.2: Comparação entre as estratégias paralelas exatas em plataformas de alto desempenho.

Ref.	Ano	Arquitetura	Redução	Seqs. (Tam. médio)	Tempo (hh:mm:ss)	*Speedup*
MSA 1.0 [74]	1989	*Single Thread*	CL	5 (60)	00:01:00	N/A
MSA 2.0 [31]	2000	*Single Thread*	CL	10 5	00:02:17 00:00:37	2,96 2,13
Helal et al. [44]	2009	MPI em Cluster (8 *cores*)	CL	2 (20) 5 (41)	00:00:00,11 02:19:41	N/A
CL-FPGA [79]	2007	FPGA	CL	4 (370,5) 4 (960,75) 5 (397) 5 (467,8)	00:00:02,33 00:00:19,6 00:05:04 00:03:28	49,0 49,0 2,00 0,07
MSA-GPU [124]	2013	GPU	CL	3 (59) 3 (417,1) 3 (453)	00:00:00,02 00:00:03,11 00:00:03,61	0,02 6,02 8,60
PFA*-DDD [89]	2006	MPI em Cluster (32 *cores*)	A-Star	5 (361,6) 5 (361,6) 5 (490,8)	00:42:00 01:56:24 11:55:12	19,5 12,6 32,0
PE2A* [37]	2013	Multicore com memória externa (12 *cores*)	A-Star	5 (361,6) 5 (490,8)	09:30:00 216:42:39	5,21 3,48

5.4 A-Star Paralelo em Outros Domínios

O algoritmo A-Star é amplamente utilizado na solução de diversos problemas, além do alinhamento múltiplo de sequências. Nesta seção, descrevemos algumas estratégias paralelas do algoritmo A-Star para outros domínios.

Zhoul *et al.* propuseram GA* [140], uma estratégia paralela usando GPUs para acelerar o A-Star. Para explorar o paralelismo da GPU, os autores usaram um grande número de filas de prioridade. Durante a etapa de detectar nós duplicados (Seção 2.3.2.1), eles

utilizaram duas abordagens diferentes. A primeira abordagem, *parallel cuckoo hashing*, é uma versão paralela do *cuckoo hashing*, algoritmo para solução de colisão de *hash*. A segunda abordagem, *parallel hashing with replacements*, utiliza estrutura de dados probabilística e é otimizada para GPUs.

Nos resultados experimentais foi utilizada uma NVidia Tesla K20c e o GA* foi testado para três problemas diferentes. Para o problema Sliding Puzzle (utilizando *15-Puzzles* e *24-Puzzles*), o GA* atingiu um speedup de aproximadamente 30x, quando comparado a uma solução *single thread* em CPU. O GA* foi testado para o problema de encontrar o menor caminho entre dois vértices em um grafo. Além disso, o GA* foi também aplicado ao problema de *Protein Design (protein folding)*. Neste caso, os resultados mostram que a fase que consumia maior período de tempo foi a de calcular a função heurística, e não a fase de busca. Essa operação foi otimizada para GPU e foi possível obter um *speedup* de 45× quando comparado a solução *single thread* em CPU. A desvantagem dessa estratégia é exigir que a maior parte dos nós do espaço de busca caibam na memória. O GA* não foi aplicado ao alinhamento múltiplo de sequências.

Sturtevant et al. [120] propuseram PEMM, um algoritmo para a busca bidirecional do A-Star, utilizando memória externa (disco). Os autores propuseram que a busca deve ocorrer em duas direções: do nó de início até o final (*forward*) e do nó final ao inicial (*backward*), com duas listas (OpenList e ClosedList) em cada uma das direções. Além disso, alguns conjuntos de nós de cada uma das listas são salvos em disco. Nos resultados experimentais, foi utilizada uma estação com dois processadores com 8 núcleos, Intel Xeon E5 2,4 GHz, 128 GB de memória RAM e 8 TB de disco. Para a solução de um *20 depth* Rubik's cube, foram necessárias 28 horas e 5 TB de disco.

Jinnai et al. [59] propuseram Abstract Zobrist hashing (AZHDA*), uma estratégia baseada em *Hash Distributed A** [61]. Nela, é utilizada a função de *hash* Zobrist, porém em alguns problemas essa função ocasiona uma transferência muito frequente de nós entre os processadores [6]. Para resolver esse problema, os autores propuseram o uso de uma função de *hash* organizada em dois níveis. A primeira função é utilizada para abstrair algumas propriedades dos nós, sendo dependente do problema no qual a solução é aplicada(*15-Puzzle, 24-Puzzle* ou MSA). Então uma segunda função de *hash* é utilizada para distribuir os nós entre as *threads*. Com a combinação dessas funções, chamada de *2-level abstract hash function*, nós que estão perto no espaço de busca tendem a ser atribuídos para o mesmo processador. Nos resultados experimentais, uma estação com 16 *cores* e 128 GB de memória RAM foi utilizada, executando 100 instâncias do *15-Puzzle* e *24-Puzzle*. Para o problema do alinhamento múltiplo de sequências, os autores utilizaram 60 instâncias do BAliBASE 3.0 além de 50 sequências geradas aleatoriamente. Os resultados mostram que o AZHDA* é mais eficiente que o HDA*. Foi relatado que o

AZHDA*, quando resolve o problema do alinhamento múltiplo é $4,6\times$ mais lento quando comparado ao problema do *24-Puzzle*. Isso ocorre pois, para o MSA, a fase de expansão dos nós exige mais recursos computacionais.

5.5 Quadro Comparativo de Estratégias A-Star

Nesta seção são comparadas as implementações do A-Star paralelo descritas nas Seções 5.2 e 5.4, Nessa comparação são considerados o tipo de arquitetura paralela utilizada, o uso de memória exerna, o uso de *hash* sensível ao contexto e o domínio que o A-Star foi aplicado.

Tabela 5.3: Comparação entre estratégias paralelas baseadas em A-Star

Nome	Ano	Estratégia Paralela	Uso de Disco	Sensível à Localidade	Domínio
PFA*-DDD [89]	2006	MPI	Não	Não	MSA
PE2A* [37]	2013	*Multithread*	Sim	Não	MSA
GA* [140]	2015	GPU	Não	Não	SPu, PF, PDe
PEMM [120]	2016	*Multithread*	Sim	Não	RC
AZHDA* [59]	2016	*Multithread*	Não	Sim	SPu, MSA

A Tabela 5.3 apresenta uma comparação dos trabalhos do A-Star paralelo. Nela, são comparadas as soluções para resolver os problemas: Alinhamento Múltiplo de Sequências (MSA), Pathfinding (PF), Sliding Puzzle (SPu), Protein Design (PDe) e Rubik Cube (RC). Com a exceção do PFA*-DDD e GA*, todas as estratégias paralelas executam em múltiplas *threads* em CPU. Duas abordagens (PE2A* e PEMM) utilizam uma memória externa para resolver instâncias maiores dos problemas. Apenas o AZHDA* utiliza funções de *hash* sensível ao contexto para distribuir os nós entre os processadores.

Capítulo 6

Previsão de Estrutura Secundária em Arquiteturas Paralelas

Nesse capítulo, serão discutidas propostas para execução de algoritmos para a previsão de estrutura secundária em arquiteturas paralelas.

Diferentemente do que acontece com a estrutura primária, não estão disponíveis muitas propostas que consideram a estrutura secundária. As soluções encontradas na literatura limitam-se aos algoritmos mais simples, para a previsão da estrutura secundária de uma única sequência de RNA. As técnicas de paralelização do algoritmo de Nussinov são discutidas na Seção 6.1. As propostas encontradas na literatura são explicadas na Seção 6.2. Por fim, na Seção 6.3, é discutido um quadro comparativo das propostas.

6.1 Técnicas de Paralelização do Algoritmo de Nussinov

Nas propostas encontradas na literatura, são explorados diferentes tipos de paralelismo que podem ser agrupados em três grupos principais: paralelismo inter-sequências, paralelismo intra-sequência e paralelismo intra-células. Esses tipos de paralelismo não são excludentes e, frequentemente, são combinados entre si.

6.1.1 Paralelismo Inter-Sequências

O paralelismo inter-sequências consiste em receber mais de uma sequência e calcular a estrutura secundária delas paralelamente. Para isso, é mantida em memória uma matriz de programação dinâmica (DP) para cada uma das sequências. Como não há dependência de dados entre elas, não há comunicação ou sincronização entre as *threads* responsáveis pelo

cálculo de cada matriz. Para lidar com sequências de tamanhos diferentes, é necessário um balanceamento de carga, pois o processamento das matrizes DP pode ter tempo diferentes.

6.1.2 Paralelismo Intra-Sequência e *Wavefront* do Algoritmo de Nussinov

O método de paralelismo Intra-Sequência consiste em utilizar múltiplas *threads* para calcular em paralelo diferentes células da matriz DP. O método de *wavefront* [105] é amplamente utilizado para paralelizar o cálculo de matrizes de programação dinâmica em geral. Este nome foi dado pelo formato de processamento das células em onda.

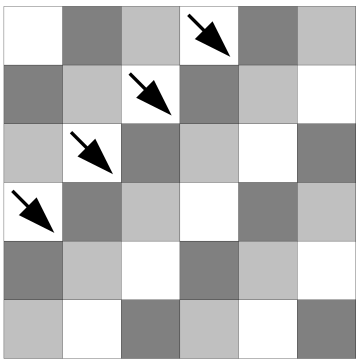
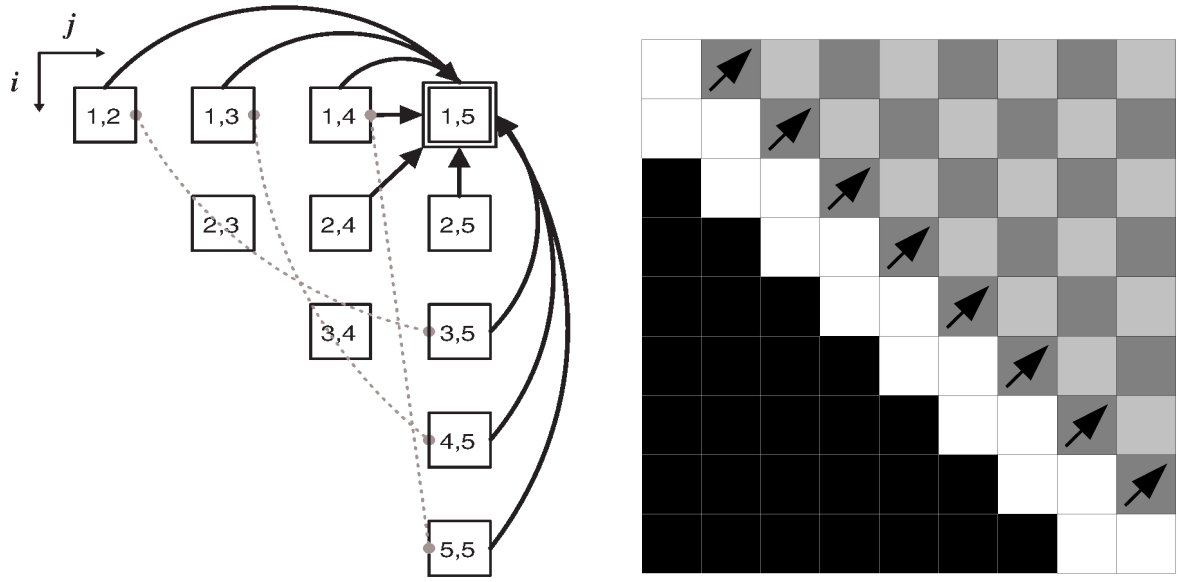


Figura 6.1: *Wavefront* de uma matriz de programação dinâmica do algoritmo de Needleman-Wunsch.

A Figura 6.1 ilustra o *wavefront* para o algoritmo de Needleman-Wunsch (Seção 2.2.1) e Smith-Waterman (Seção 2.2.2). No começo, apenas a primeira célula em branco, no topo à esquerda, pode ser processada. Depois, duas células podem ser processadas em paralelo. Esse processo continua e o paralelismo máximo é atingido quando a antidiagonal principal é processada. Após isso, o paralelismo volta a diminuir e ao final apenas uma célula, no canto inferior direito, pode ser processada.

A Figura 6.2 ilustra (a) a dependência de dados de uma célula com $(i, j) = (1, 5)$; e (b) o *wavefront* do algoritmo de Nussinov. O algoritmo de Nussinov possui uma dependência complexa de dados. Neste exemplo, a célula possui como dependência as células adjacentes $(1, 4)(2, 4)$ e $(2, 5)$, mas também deve aplicar a regra de *multibranch* para células que não são adjacentes $(1, 4) + (5, 5)$, $(1, 3) + (4, 5)$ e $(1, 2) + (3, 5)$.

O *wavefront* do algoritmo está ilustrado na Figura 6.2b. Nela, as células em preto não são processadas por não fazer parte do espaço de busca. As células em branco são inicializadas como 0 pelo algoritmo (Seção 3.2.1). Como as células da matriz de programação dinâmica possuem como dependências as células das diagonais anteriores,



(a) (b)

Figura 6.2: (a) Dependência de dados de uma célula $(1, 5)$ do algoritmo de Nussinov [57] e (b) Wavefront do algoritmo de Nussinov.

células que estão na mesma diagonal podem ser processadas em paralelo. Desta forma, o número máximo de células que podem ser calculadas é obtido na primeira diagonal processada. Esse número vai reduzindo a cada diagonal processada e no final apenas uma célula pode ser processada, no canto superior direito.

6.1.3 Paralelismo Intra-Células

O paralelismo intra-células consiste em utilizar diferentes *threads* para calcular a mesma célula da matriz de programação dinâmica. O termo de *multibranch* introduz uma complexa dependência de dados no dobramento da estrutura secundária. Para aplicar a regra de *multibranch*, considerando uma sequência de tamanho n, a última célula calculada da matriz, $(0, n)$, deve ler $n - 1$ células da linha 0 e somar esses valores a $n - 1$ células da coluna n, porém apenas o maior valor dessas somas deve ser armazenado, sendo possível utilizar múltiplas *threads* para obter esse valor.

No entanto, as células das primeiras diagonais possuem poucos termos de *multibranch* que podem ser calculados em paralelo. Por isso, utilizar exclusivamente o paralelismo intra-células levaria a uma execução praticamente serial nas primeiras diagonais. Não conhecemos na literatura alguma estratégia que utilize exclusivamente o paralelismo intra-células, sendo normalmente combinado com os outros paralelismo para um melhor desempenho.

6.2 Previsão de Estrutura Secundária de uma Sequência em Arquiteturas de Alto Desempenho

6.2.1 Dobramento de uma Sequência em FPGA - Jacob et al.

Jacob et al. [57] propuseram uma estratégia para implementar o algoritmo de Nussinov (Seção 3.2.1) em FPGAs (Seção 4.2). Inspirados no problema de parentetização de *strings*, eles utilizaram uma técnica de "*middle serialization*" [24], para criar o conjunto equivalente de equações de recorrência de Nussinov (Equação 3.1), produzindo a Equação 6.1.

$$D_{i,j,k} = \max \begin{cases} s(S_1(i), S_2(j)) & \text{se } j - i = 1 \\ \\ D_{i+1,j,k} & \\ D_{i,j-1,k} & \\ D_{i+1,j-1,k} + s(S_1(i), S_2(j)) & \text{se } k = 1 \\ D_{i,j,k+1} & \\ D_{i,i+k,1} + D_{i+k+1,j,1} & \\ D_{i,j-k,1} + D_{j-k+1,j,1} & \\ \\ D_{i,j,k+1} & \\ D_{i,i+k,1} + D_{i+k+1,j,1} & \text{Caso contrário} \\ D_{i,j-k,1} + D_{j-k+1,j,1} & \end{cases} \tag{6.1}$$

Na equação 6.1, foi introduzida uma nova dimensão k à matriz de programação dinâmica, tal que $1 \leq k \leq \lfloor \frac{(j-i)}{2} \rfloor$. O escore da melhor estrutura está presente na célula $D(1, N, 1)$.

A vantagem de utilizar essa equação de recorrência é que ela não possui o termo de *multibranch*, sendo as células da matriz de programação dinâmica de 3 dimensões possuem uma dependência de dados uniforme, isto é, cada célula da matriz realiza o mesmo número de operações de leitura.

Os autores propuseram sua solução dividindo as operações em matrizes 3 x 3 e também implementaram um *pipeline* para eliminar algumas verificações lógicas e otimizar o circuito. Além disso, foram criados *arrays* sistólicos baseados em 2 métodos de parentetização de strings. Foram chamados de GKT e GJQ e cada um possui capacidade de processamento para sequências de 34 e 62. Múltiplos *arrays* sistólicos são criados para dobrar simultaneamente diversas sequências, em uma abordagem inter-sequências.

A estratégia proposta foi implementada implementada em VHDL e comparada com um programa próprio, escrito em C, que implementa o algoritmo de Nussinov em CPU. Para esse teste, foi utilizada uma estação Intel Core 2 Duo de 3 GHz. Para a implementação em FPGA, foi utilizada uma placa FPGA Virtex-II 6000 e o circuito utilizado possuía 70 MHz. Mil sequências sintéticas, geradas aleatoriamente, com tamanho de 30 a 60 nucleotídeos foram utilizadas. O tempo total em CPU foi de 68.021 ns e em FPGA foi de 1.715 ns, obtendo um *speedup* de 39×.

Esses resultados mostram um ganho de velocidade, porém, os testes utilizaram sequências de tamanho limitado. Consequentemente, o tempo de execução em CPU e FPGA foi bastante reduzido na ordem de microsegundos.

6.2.2 Dobramento de uma Sequência em GPU - Ritz et al

Rizk [108] et. al apresentaram um algoritmo para descobrir a estrutura secundária de uma única sequência (Seção 3.2) acelerando o processo com o uso de GPU. O algoritmo utilizado é uma variante do algoritmo de Nussinov (Seção 3.2.1) que não utiliza apenas a contagem de pares de bases, mas utiliza um modelo de minimização de energia livre, em uma única sequência. Neste caso a dependência de dados permanece quase inalterada, porém é utilizada uma matriz de programação dinâmica a mais, para armazenar em qual estado se encontra uma determinada célula.

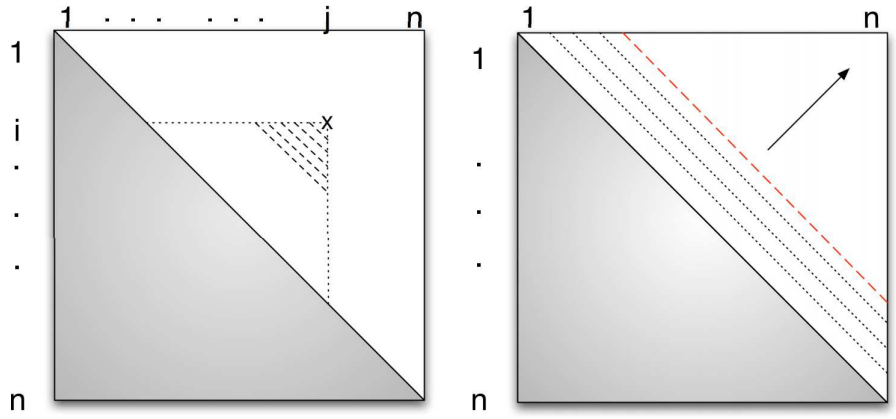


Figura 6.3: Estratégia de Rizh et al. A dependência de dados de uma única célula é apresentada à esquerda e à direita são apresentadas células que podem ser calculadas em paralelo (*wavefront*) [108].

A paralelização do algoritmo é de intra-sequência, conforme mostra a Figura 6.3. À esquerda é mostrada a dependência de dados de uma única célula, que possui como dependência três células anteriores (Equação 3.1), e também toda a linha e coluna da

célula. Desta forma, as células que podem ser calculadas em paralelo são as diagonais da matriz de programação dinâmica, como está mostrado na parte da direita da figura.

Neste trabalho também é explorado um nível de paralelismo de uma granularidade maior, de inter-sequências, permitindo que sejam calculadas múltiplas tarefas independentes. Desta forma, múltiplas sequências, com baixo consumo de memória, são atribuídas para a mesma GPU para serem dobradas em paralelo. Com esta granularidade foi criada uma abordagem multi-GPU, de forma que são distribuídas tarefas independentes entre várias GPUs.

Nos testes realizados, é citado que a procura de um micro RNA em um genoma completo requer que sejam conhecidas as estruturas secundárias de milhões de sequências com comprimento de 120 nucleotídeos [119]. Para a validação da implementação proposta, são utilizadas 40.000 sequências com este tamanho de sequência. O tempo em GPU é comparado com um programa sequencial amplamente utilizado chamado Unafold [78], porém eram executadas em paralelo oito chamadas ao programa.

Os melhores resultados foram obtidos com a placa GTX 280, uma das melhores disponíveis na época, conseguindo *speedup* de $17,0$x utilizando-se uma GPU e sendo comparada a um core de CPU e $33,1$x ao se utilizar duas GPUs. Apesar do tempo ser reduzido, a acurácia se manteve, pois foram utilizadas as mesmas regras termodinâmicas utilizadas pelo programa em CPU.

6.2.3 Dobramento de uma Sequência utilizando CPU e GPU - Lei et al.

Lei *et al.* [71] apresentam uma abordagem híbrida para a previsão de uma estrutura secundária de um único RNA (Seção 3.2). Em abordagens que utilizam GPU, é desejável também se aproveitar a CPU durante a execução do problema, afinal os sistemas possuem *cores* de CPU que muitas vezes podem ficar ociosos enquanto a GPU realiza todo o trabalho. Nesta abordagem, são criados dois módulos, um para execução em CPU *multicore*, que faz uso de instruções vetoriais para obter um melhor desempenho e uma arquitetura em GPU também é proposta. Os módulos são combinados e escalonados de uma forma simples de maneira a distribuir tarefas quando o dispositivo (CPU ou GPU) fica ocioso.

Para a implementação em uma CPU *multicore*, são utilizadas múltiplas *threads* e instruções vetoriais, onde elementos da matriz de programação dinâmica são simultaneamente adicionados. Essa matriz representa a junção de melhores subestruturas e é semelhante à ultima parte da Equação 3.1. As instruções vetoriais otimizam essa operação, adicionando simultaneamente os elementos em um registrador de 128 bits.

Diversos esquemas de paralelismo são utilizados para a implementação do algoritmo em GPU, apresentando uma estrutura de paralelismo hierárquico. Esta abordagem é muito semelhante ao paralelismo apresentado na Seção 6.2.2, onde dentro de um *kernel*, as células da matriz de programação dinâmica são calculadas em paralelo. Da mesma forma, um paralelismo de granularidade mais grossa é utilizado para calcular simultaneamente diversas sequências, utilizando uma abordagem de paralelismo inter-sequências.

Para os testes, foram utilizados quatro grupos com 20.000 sequências para serem calculados em paralelo, explorando o paralelismo intra-sequência. Cada grupo possuía sequências que variavam entre o comprimento entre 68 e 221. A máquina era equipada com um Intel Xeon E5620 Quad 2,4 GHz CPU, 24 GB RAM e a GPU era uma Geforce GTX580 com o CUDA *toolkit* 4.0.

Os resultados mostram um *speedup* de $15,93\times$ sobre uma implementação otimizada com instruções vetoriais, utilizando apenas CPU e uma redução de 16% em tempo de execução em relação a uma implementação exclusiva de GPU.

6.2.4 Dobramento de uma Sequência em CUDA - Li et al.

Li et. al [72] propuseram uma estratégia em GPUs utilizando CUDA (Seção 4.4.1) para acelerar o algoritmo Nussinov (Seção 3.2.1) e prever a estrutura secundária de uma sequência de RNA. Nesta estratégia, a matriz de programação dinâmica não é processada em diagonais, mas sim em blocos de diagonais.

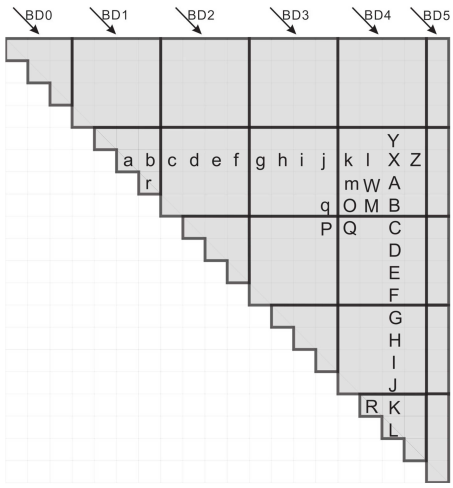


Figura 6.4: Processamento da matriz de programação dinâmica em blocos [72].

A Figura 6.4 ilustra um exemplo do processamento de uma matriz de uma sequência de tamanho 20. Nela, está ilustrado a divisão em blocos da matriz, além de mostrar a dependência de dados da regra de *multibranch* para uma das células da matriz, chamada de X. Dentro dos blocos, as células que estão na mesma diagonal são calculadas em

paralelo. Na figura, os blocos possuem 7 diagonais e são necessárias 7 iterações para se calcular todo o bloco.

Porém, antes de iniciar o processamento do bloco, são realizados em paralelo diversas operações de *multibranch* para o bloco, em uma abordagem de paralelismo intra-células. No exemplo da figura 6.4, considere o bloco que contém a célula X. Suponha que as células A e B ainda não foram calculadas. Porém, as células C e c, D e d, já foram calculadas por pertencerem a blocos anteriores e devem ser utilizadas para a regra de multibranch de X. Então, antes de processar as diagonais de um bloco, as células que utilizam operações de *multibranch* e pertencem a blocos anteriores são calculadas e armazenadas temporariamente. Apenas o maior valor entre as regras de *multibranch* precisa ser armazenado e, portanto, os outros são descartados.

Nesta trabalho, foram criadas duas versões. A primeira versão armazena totalmente a matriz de programação dinâmica 2D. Porém o algoritmo de Nussinov utiliza apenas metade dessa matriz, conforme ilustrado na Figura 3.4. Na segunda versão, apenas metade da matriz é armazenada, porém nesse caso uma função de mapeamento deve ser utilizada para acessar a memória.

Nos resultados experimentais foi utilizada uma estação com um Intel i7x980 com 3,33 GHz CPU, 12 GB de RAM. A estação também possuía uma GPU NVidia Tesla C2050. Os resultados experimentais mostram que para sequências de tamanho 26.000, a primeira versão executa em 23,9s, enquanto a segunda versão executa em 26,7s. Porém, como a segunda versão utiliza menos memória, ela é capaz de dobrar sequências de 37.000 nucleotídeos, enquanto a primeira versão não consegue. Os autores concluem que a versão em GPU é mais rápida que o código executando em CPU, utilizando a primeira versão do algoritmo quando há memória disponível no sistema.

6.2.5 Dobramento de uma Sequência com Intel Xeon Phi - Palkowski e Bielecki

Palkowski e Bielecki [103] propuseram uma estratégia paralela para acelerar a execução do algoritmo de Nussinov (Seção 3.2.1) utilizando diversos aceleradores Intel Xeon Phi (Seção 4.3). Através da análise da dependência de dados da equação de recorrência de Nussinov, eles criaram um conjunto de equações equivalentes, mas que processam a matriz de programação dinâmica por blocos, referido no trabalho como *tiles*, e que possuem um padrão de acesso a memória mais uniforme.

A estratégia da Seção 6.2.4 também utiliza blocos. Em ambas as abordagens, o objetivo é agrupar as células que realizam operações de leitura em regiões de memória próximas. Porém, a estratégia da Seção 6.2.4 utiliza blocos para aproveitar a memória

compartilhada das GPUs (Seção 4.4.2), enquanto a abordagem desta seção busca agrupar para utilizar instruções vetoriais e melhor aproveitamento de memória *cache* da CPU e Intel Xeon Phi.

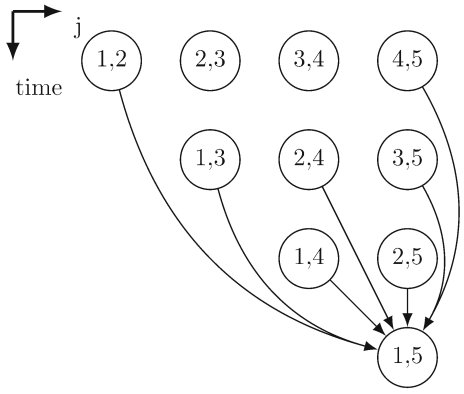


Figura 6.5: Escalonamento das células da matriz de Programação Dinâmica de Nussinov produzido com o *loop skewing* [103].

Para a paralelização do algoritmo, foi utilizada uma técnica chamada de paralelização chamada de *loop skewing transformation* [136]. Esta técnica é utilizada para mapear *loops* aninhados em dois ou mais *loops*, de forma que o *loop* externo seja serial e os internos possam ser paralelos. O índice do *loop* externo é uma combinação de índices dos originais.

A Figura 6.5 ilustra o escalonamento das células da matriz DP produzido com o *loop skewing*. Nela, células que estão na mesma linha podem ser processadas em paralelo. A dependência de dados da célula $(i, j) = (1, 5)$ é mostrada na figura. Essa técnica de paralelização detectou que as diagonais da matriz DP podem ser processadas em paralelo, criando um padrão de *wavefront* de paralelismo (Seção 6.1).

Para avaliar a estratégia, foi utilizada uma máquina com dois processadores Intel Xeon E5-2699 v3, 3,2 GHz e 32 *cores* cada um e 128 GB de RAM. Essa estação era equipada com quatro aceleradores Intel Xeon Phi 7120P, 1,238 GHz e 61 *cores*. Um código serial do algoritmo de Nussinov foi implementando em C. O compilador TRACO [104] é utilizado para gerar o código *tiled* serial e paralelo. Posteriormente, esses códigos gerados são compilados com o compilador Intel C++ versão 17.0.1.

Para os testes, foram utilizadas sequências aleatórias de tamanho 2.200 e 5.000, além de sequências reais. O código serial original em CPU é comparado ao código *tiled* em CPU, reduzindo o tempo total de excução de 12,28s para 8,25s e de 334,32s para 225,23s, para as sequências de tamanho 2.200 e 5.000, respectivamente. Ao se utilizar o Intel Xeon Phi com 244 *threads* o tempo foi reduzido para 3,72s e 37,75s. Porém os melhores resultados são obtidos ao se utilizar 64 *threads* em CPU, com um tempo reduzido para 0,37s e 10,50s.

Além disso, também foi comparado o ganho de velocidade utilizando o compilador PluTo [5] para gerar o código paralelo. Nesta comparação, os resultados mostram que o *speedup* do compilador TRACO é superior para todos os tamanhos de sequências e números de *threads*. Eles atribuem o ganho de velocidade ao fato que o compilador PluTo não consegue paralelizar todos os *loops* do algoritmo de Nussinov. Os autores concluem que é possível criar uma estratégia paralela *tiled* para o algoritmo de Nussinov e que a abordagem de paralelização, para os experimentos realizados, é melhor que as disponíveis atualmente.

6.3 Quadro Comparativo

A Tabela 6.1 apresenta as propostas paralelas para previsão da estrutura secundária de uma única sequência em plataformas de alto desempenho. Até onde sabemos, não existe na literatura trabalho que utilize o paralelismo em arquiteturas de alto desempenho no alinhamento estrutural de sequências de RNA (Seção 3.3).

A Tabela 6.1 mostra que das 5 propostas analisadas, em 3 delas o paralelismo é explorado ao se dobrar um número muito alto de sequências em paralelo. Nessas estratégias são utilizadas de 1.000 a 40.000 sequências em paralelo, com tamanho de 34 a 221. Nas outras 2 estratégias, o paralelismo é utilizado para acelerar a obtenção da estrutura secundária de apenas uma sequência, com tamanho entre 2.200 e 14.000.

Os trabalhos analisados possuem uma variedade grande de arquiteturas alvo, tais como FPGAs, CPU, GPU, CPU-GPU e Intel Xeon Phi. As arquiteturas de *hardware* reconfigurável (FPGAs) utilizam sequências pequenas, quando comparada às outras estratégias. Os melhores *speedups* e menores tempos de execução são obtidos ao se utilizar GPUs, realizando o dobramento de uma sequência de 14.000 nucleotídeos em apenas 4,5s.

Tabela 6.1: Comparação entre as estratégias paralelas para previsão de estrutura secundária em plataformas de alto desempenho.

Ref.	Ano	Arquitetura	Tipo de Paralelismo	Seqs. (Tam. médio)	Tempo (s)	*Speedup*
[57]	2008	FPGA	Inter-sequências	1.000 (34) 1.000 (62)	<0,01 <0,01	14,94 49,66
[108]	2009	CPU (8 *cores*) 1xGPU 2xGPU	Inter-sequências e Intra-sequência	40.000 (120) 40.000 (120) 40.000 (120)	40,8 18,9 9,5	7,93 17,10 33,10
[71]	2012	CPU-GPU	Inter-sequências e Intra-sequência	20.000 (120) 20.000 (154) 20.000 (221)	5,67 13,87 56,38	15,93 11,80 6,75
[72]	2014	GPU	Intra-sequência e Intra-células	1 (6.000) 1 (14.000)	0,6 4,5	606,16 1.335,3
[103]	2017	CPU (64 *cores*) CPU (64 *cores*) 4xIntel Xeon Phi 4xIntel Xeon Phi	Intra-sequência	1 (2.200) 1 (5.000) 1 (2.200) 1 (5.000)	0,37 10,50 3,72 37,75	33,19 31,84 3,30 8,85

Parte II

Contribuições

Capítulo 7

PA-Star: Estratégia Paralela com *Hash* Sensível à Localidade para o Alinhamento Múltiplo Exato de Sequências

O PA-Star [125, 126] é a primeira contribuição desta tese. A principal ideia da estratégia paralela A-Star (Seção 2.3.2.1) é executar em paralelo tanto a fase de busca como a de expansão do algoritmo A-Star para a solução do alinhamento múltiplo exato de sequências. Os principais desafios enfrentados na confecção do PA-Star consistiram no padrão irregular de expansão dos nós e do crescimento muito grande das listas *OpenList* e *ClosedList*. Para abordar o primeiro desafio, propusemos um particionamento do espaço de busca A-Star e uma estratégia de alocação de *threads* sensível à localidade [126]. Para aumentar o espaço de memória para as listas, propusemos o uso de disco, quando a memória RAM não fosse suficiente [125].

O objetivo deste capítulo é detalhar o projeto do PA-Star. O projeto da estratégia, o algoritmo paralelo e suas otimizações são detalhados na Seção 7.1. Os ambiente de testes e os resultados experimentais são descritos na Seção 7.2. Por fim, na Seção 7.3 é apresentada uma discussão sobre o PA-Star.

7.1 Projeto do A-Star Paralelo

O PA-Star é uma estratégia paralela que obtém o alinhamento exato ótimo com a redução do espaço de busca baseado no algoritmo A-Star (Seção 2.3.2.1). O PA-Star possui dois objetivos. Primeiro, ele reduz o tempo de execução utilizando uma função de *hash* sensível à localidade para distribuir os nós do A-Star entre as *threads*. Segundo, ele

busca resolver conjuntos de sequências que exigem uma grande quantidade de memória combinando memória RAM e disco. Esses dois objetivos estão em conflito entre si, afinal o uso em disco claramente aumenta o tempo de execução. Portanto, se existir memória RAM disponível, o PA-Star não utiliza disco. Porém, se o PA-Star observar que está sem memória RAM disponível, a estratégia começa a utilizar memória RAM e disco.

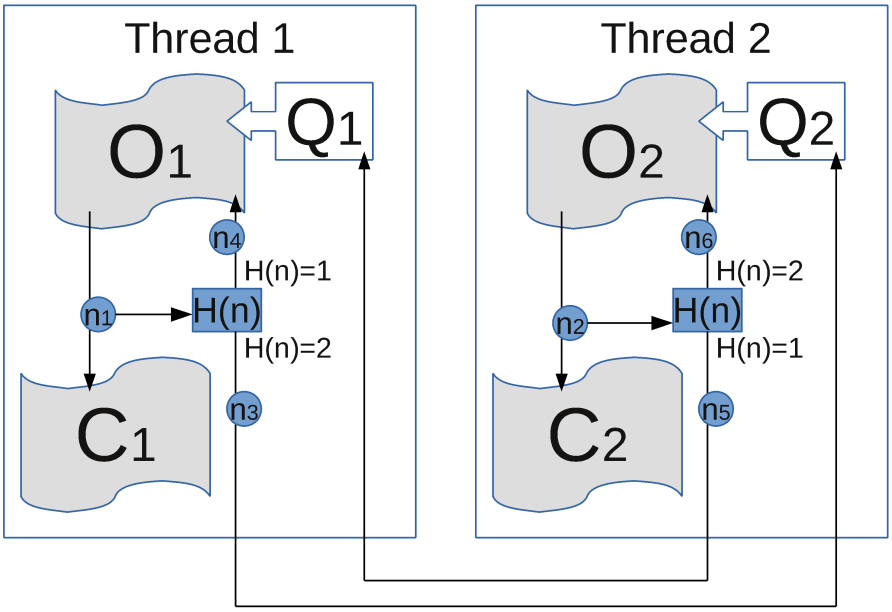


Figura 7.1: Diagrama da estratégia PA-Star utilizando 2 *threads*.

A Figura 7.1 apresenta uma visão geral do PA-Star com 2 *threads*. No PA-Star, os nós contidos na *OpenList* são expandidos em paralelo e as listas do A-Star são gerenciadas localmente por t *threads*. Portanto, cada thread t_i possui a sua própria $OpenList_i$, $ClosedList_i$ e a fila q_i (Figura 7.1). Os nós que compõem o espaço de busca são distribuídos entre as *threads* utilizando uma função de *hash* sensível à localidade.

Todos os nós do espaço de busca possuem um campo único de coordenada que os identificam. O nó inicial possui coordenada $C_i = (0, 0, ..., 0)$ e o nó final possui coordenada $C_f = (l_1, l_2, ..., l_k)$, onde k é o número de sequências e l_j é o comprimento da j-ésima sequência. Para mapear os nós que serão processados pela *thread* t_i, armazenados na $OpenList_i$ e $ClosedList_i$, utiliza-se a Equação 7.1. Nesta equação, H é a função de *hash* sensível à localidade, $(C_1, C_2, ..., C_k)$ é a coordenada do nó, S é um constante de *shift-right*, t é o número total de *threads* e t_i é o identificador da *thread*. A operação *mod* é usada para mapear o resultado final do *hash* em um valor, tal que $0 \leq t_i < t$. Nós finais são uma exceção à essa regra e são escritos em todas as *OpenLists*, conforme explicado

ao final desta seção.

$$(H(C_1, C_2, ..., C_k) \gg S) \bmod t = t_i \tag{7.1}$$

Se a *thread* t_i expande o nó m, é desejável que o nó m seja adicionado à $OpenList_i$, reduzindo o *overhead* de comunicação e aproveitando um melhor uso de *caches*. Para obter isso, é proposto o uso de (a) funções de *hash* sensíveis à localidade, que preservam a localidade e tendem a mapear os nós expandidos para a mesma *thread*; e (b) operações *shift-right*, onde os bits menos significativos do *hash* são descartados. No PA-Star, utilizamos duas funções H: a função Sum, que realiza a soma de todos os números na coordenada; e a função Zorder, que é baseada nas curvas Z-Order [86]. A função Zorder foi escolhida porque ela preserva a localidade e é computacionalmente simples de ser calculada, pois consiste em operações de *shift* para intercalar os bits dos números da coordenada.

A Figura 7.1 ilustra o funcionamento do PA-Star para 2 *threads*. Conforme mencionado anteriormente, cada *thread* i possui uma *OpenList* (O_i), uma *ClosedList* (C_i) e uma fila (Q_i). As *threads* expandem os nós em paralelo (n_1 e n_2), removendo os nós da *OpenList* correspondente e adicionando-os às respectivas *ClosedList*. Durante a expansão, a função de *hash* sensível à localidade (H) (Equação 7.1) é utilizada para determinar em qual $OpenList_i$ o nó vizinho deve ser inserido. Nós que pertencem à mesma *thread* são inseridos diretamente na respectiva *OpenList* (n_4 e n_6), enquanto nós que pertencem à *thread* j são inseridos na fila Q_j (n_3 e n_5).

7.1.1 Algoritmo PA-Star

O PA-Star é dividido em 2 fases alternadas executadas em paralelo: *search_step* e *verify_end_condition*. A etapa *search_step* implementa a etapa de busca e expansão do algoritmo A-Star (Seção 2.3.2.1, Algoritmo 2), das linhas 7 a 16 e a etapa *verify_end_condition* implementa a linha 6 deste algoritmo.

O PA-Star é executado da seguinte forma: inicialmente, a *thread* t_0 executa as linhas 1 a 4 (Algoritmo 8). A função de *hash* H é aplicada ao nó inicial N_i e o id h é obtido (linha 1). O nó inicial N_i é colocado na *OpenList* da *thread* h (linha 2). A condição de parada *end_condition* e o valor de custo f até o nó final n são inicializados como falso e ∞, respectivamente (linhas 3 e 4). Depois, t *threads* executam as duas etapas do algoritmo em paralelo (linhas 5 a 9).

O *search_step* do PA-Star é descrito no Algoritmo 9. Todas as *threads* executam uma versão modificada do algoritmo A-Star. Enquanto o contador de *threads*, *threads_counter*, é menor que o número total de *threads*, (linha 3), a *thread* t_i consome a fila q_i, removendo

Algoritmo 8 *PA-Star*(N_i, N_f)

```
1: h ← hash(N_i) /* Obtém a thread de destino */
2: OpenList_h ← N_i /* Adiciona o nó inicial à OpenList */
3: end_condition ← false
4: n ← ∞
5: repeat
6:    /* Em paralelo i = 1 a t */
7:    n ← search_step(i, N_f) /* Executa a etapa de busca para thread i */
8:    end_condition ← verify_end_condition(i) /* Condição de parada */
9: until end_condition == false
10: return n
```

Algoritmo 9 *search_step*(i, N_f)

```
1: threads_counter ← 0
2: final_node ← ∞
3: while threads_counter < threads_num do
4:    /* Início da etapa de busca */
5:    consume_queue(q_i) /* Insere todos os nós da fila q_i na OpenList_i */
6:    current ← OpenList_i.get_lowest_f() /* Remove o nó de melhor prioridade da
      OpenList_i */
7:    if current.g ≤ ClosedList_i.find_g(current) then
8:       /* Se um nó de melhor prioridade ainda não foi encontrado */
9:       ClosedList_i ← ClosedList_i ∪ current /* Adiciona o nó na ClosedList_i */
10:      if current ∈ N_f then
11:         /* Se o nó é um nó final*/
12:         threads_counter++ /* Aumenta os contadores de threads */
13:         final_node ← current /* Salva o nó final */
14:         process_final_node(current, threads_counter) /* Verifica condição de pa-
            rada */
15:      else
16:         for neighbor in neigh(current) /* Fase de expansão */ do
17:            h ← hash(neighbor) /* Obtém a thread destino */
18:            q_h ← q_h ∪ neighbor /* Adiciona o nó na fila da thread h */
19:         end for
20:      end if
21:   end if
22:   /* Fim da etapa de busca */
23: end while
24: return final_node
```

todos os nós da q_i e os adicionando à $OpenList_i$ (linha 5). Depois disso, o nó com menor prioridade f é removido da $OpenList_i$ (linha 6).

Se o nó removido da $OpenList_i$ não é um nó final, ele é adicionado à $ClosedList_i$ (linha 9) e expandido (linhas 16 a 19). Durante a fase de expansão, a função de *hash* sensível à localidade é utilizada para determinar em qual lista o nó vizinho deve ser adicionado (linha 17). Nós que pertencem à *thread* t_h são inseridos na fila q_h (linha 18).

Quando o nó expandido é final (N_f) (linha 10), este nó só deve ser considerado o resultado ótimo se tiver a menor prioridade f entre todos os nós das *OpenLists* de todas as *threads*. Essa verificação foi otimizada de maneira a dividir a verificação da condição de parada em duas fases (assíncrona e síncrona), procurando diminuir o *overhead* de comunicação das *threads*. Na fase assíncrona, o contador de *threads* $threads\_counter$, é incrementado e a função $process\_final\_node$ (linha 14) é chamada. Nesta função, se o valor de f do nó final é o menor entre todos os nós da *thread*, o nó N_f é inserido em todas as *OpenLists* e o nó N_f é salvo em uma variável global.

Todas as *threads* continuam executando a etapa de busca $search\_step$ e quando a *thread* t_i expande o nó N_f de novo, isso significa que o nó N_f tem o menor valor de f entre todos os nós da $OpenList_i$ e então o contador $threads\_counter$ é incrementado (linha 12). Quando $threads\_counter$ é igual a $threads\_num$, todas as *threads* expandiram o nó N_f, então este nó é retornado como uma possível resposta (linha 24). Porém, outros nós com menor prioridade f podem existir em outras filas q_i ou podem ter sido inseridos em alguma *OpenList* depois que a *thread* expandiu o nó N_f. Para garantir que o nó N_f é o resultado ótimo, todas as *threads* vão para a outra etapa.

Na $verify\_end\_condition$ (Algoritmo 8, linha 8), as *threads* não expandem mais nenhum nó, uma variável booleana em memória compartilhada b_s recebe o valor verdadeiro e todas as *threads* são sincronizadas. Depois, todas as *threads* consomem suas filas e cada *thread* t_i verifica se $f(N_f) < OpenList_i.lowest\_f()$ é verdadeiro. Ou seja, verifica se a possível resposta possui melhor prioridade do que todos os nós da $OpenList_i$. Senão, b_s recebe falso. Todas as *threads* são sincronizadas novamente. Depois, se b_s é verdadeiro, então o nó N_f é o resultado ótimo e $verify\_end\_condition$ retorna verdadeiro e o algoritmo retorna N_f como resposta (Algoritmo 8, linha 10). Caso contrário existe um ou mais nós em alguma *OpenList* possui um melhor valor f e precisam ser expandidos antes que seja possível garantir que N_f é o resultado ótimo. Para reiniciar a etapa de busca, o nó N_f é inserido na *OpenList* da *thread* que o expandiu pela primeira vez, a função $verify\_step$ retorna falso e todas as *threads* voltam à etapa de busca $search\_step$.

Além disso, algumas outras condições especiais devem ser verificadas. Primeiro, quando a *thread* t_i chama $consume\_queue$, mas a $OpenList_i$ está vazia, ela precisa esperar por novos nós em q_i. Outra condição especial pode ser encontrada na função

process_final_node. Se uma *thread* expande um nó final N_{f2}, e um nó N_{f1} já foi expandido, é necessário verificar se $f(N_{f2}) < f(N_{f1})$. Se isso for verdadeiro, significa que uma resposta melhor foi encontrada enquanto as *threads* consumiam as suas *queues* e expandiam as *OpenLists*. Neste caso, a verificação de N_{f1} como possível resposta deve ser cancelada, e o processo deve ser reiniciado para verificar se N_{f2} é uma possível resposta. Para isso, o contador de *threads threads_counter* é reiniciado para 1 e o nó N_{f2} é inserido em todas as *queues*. Após isso, todas as *threads* iniciam a etapa de busca *search_step*.

Antes de iniciar o PA-Star, algumas vezes é necessário efetuar operações para a função heurística h. No caso do alinhamento múltiplo de sequências, utilizamos a função $h_{2,all}$ (Seção 2.3.2.2), que exige a comparação ótima de todos as sequências par-a-par. Com a divisão de verificação de parada em duas etapas, *overhead* de sincronização entre as *threads* é reduzido, interrompendo a etapa de busca e sincronizando as *threads* apenas se uma possível resposta for encontrada.

7.1.2 Projeto da Estrutura de Dados OpenList

Um dos desafios de se implementar o algoritmo A-Star é o projeto da estrutura de dados da *OpenList*. Os nós na *OpenList* possuem dois campos diferentes que devem ser utilizados para ordenação: o valor f, representado por um inteiro e o valor da coordenada, representada por n inteiros, onde n é o número de sequências. O algoritmo A-Star precisa remover o nó com o menor valor f entre todos os nós da lista (Algoritmo 2, linha 8). Esta operação normalmente é implementada por filas de prioridade. Ele também precisa encontrar um nó utilizando as coordenadas (Algoritmo 2, linha 9), que é normalmente implementado com a ajuda de funções ou tabelas de *hash*.

Niewiadomski et al. [89] implementaram a *OpenList* em uma abordagem chamada dicionário, utilizando duas estruturas de dados: uma fila de prioridade e uma *hash table*. O problema óbvio desta abordagem é que a memória utilizada para representar os nós é duplicada.

No PA-Star, nós propusemos uma nova solução para a implementação da *OpenList*: o uso de Multi-index. A biblioteca Boost C++ (`http://www.boost.org`) permite uma classe de template multi-index, que permite a construção de estruturas que possuem mais de um índice, com diferentes regras de ordenação e acesso.

Nossa *OpenList* Multi-index, baseada na Boost C++, possui dois índices: (a) o valor f do nó; e (b) a coordenada. Foi também implementada uma *OpenList* com a abordagem de dicionário utilizando a biblioteca *Standard Template Library* (STL) [83]. Para a comparação entre as abordagens, utilizamos uma estrutura *priority_queue* da STL para operar no campo f e um *STL map* para operar no campo coordenadas. Em ambas as

implementações, a estrutura de dados que implementa a *ClosedList* é um *STL map*, pois ela não exige operações no campo f dos nós.

7.1.3 Uso de *Templates*

Cada nó espaço de busca possui uma coordenada única que o identifica e é representada por n inteiros, onde n é o número de sequências. Uma alteração no número de sequências causa um grande impacto no desempenho, já que isso modifica o número de iterações nos *loops* executados. Por isso, algumas soluções na literatura [79, 89] são otimizadas para um número específico de sequências. Uma técnica comum é criar códigos diferentes e estruturas de dados para cada número específico de sequências, normalmente entre 4 e 8 sequências

Obviamente, não é apropriado criar e manter diferentes conjuntos de funções e estruturas de dados para cada um dos números de sequências desejados. Uma possível solução para esse problema é usar memória dinâmica, alocando nós na memória de acordo com o número de sequências. A principal desvantagem em se utilizar a memória dinâmica é o aumento no consumo da memória. Por exemplo, para 4 sequências, utilizando ponteiros de 64 bits e valores inteiros de 16 bits para as coordenadas, a representação com memória dinâmica exige 2× mais espaço que a representação com *templates*.

Portanto, preferimos o uso de *templates* na solução. Eles permitem que o compilador crie variantes das estruturas de dados, classes e funções usando a mesma especificação de código. Para usar os *templates*, foi criada uma macro que permite especificar para o compilador a quantidade de sequências que desejamos comparar e o compilador utiliza essa macro para criar código especializado para cada variante. Para comparar o desempenho, também implementamos uma estratégia utilizando alocação dinâmica.

7.1.4 Módulo Disk-Assisted PA-Star (DAPA)

Conforme ilustrado na Seção 2.3.2.1, um dos grandes desafios do algoritmo A-Star é que ele frequentemente utiliza uma quantidade substancial de memória. O nosso módulo Disk-Assisted PA-Star (DAPA) é projetado para superar os limites de capacidade da memória RAM, utilizando o disco sempre que a memória RAM exceda um valor definido de *threshold*. Resultados empíricos mostram que o tamanho da *ClosedList* é consideravelmente maior que a *OpenList* na maioria das execuções e por isso optamos por mover parte da *ClosedList* para o disco, sempre que o limite de *threshold* é atingido.

Para definir quais partes da *ClosedList* devem ser transferidas para o disco, foi proposto o conceito de região. Uma região ativa é um subconjunto de nós que estão sendo processados pela *thread* t_i em uma iteração do algoritmo PA-Star. A região ativa da *th-*

read t_i contém os nós de maior prioridade entre os nós da *thread* t_i. As outras regiões são chamadas de regiões inativas. Nesta proposta, os nós que estão em regiões inativas não estão sendo processados e podem ser colocados em disco. Na próxima iteração, a região inativa pode se tornar ativa e neste caso os nós são lidos do disco para a RAM.

No projeto básico do algoritmo PA-Star (Seção 7.1.1), cada *thread* t_i processa exatamente uma região e possui uma $OpenList_i$ e uma $ClosedList_i$. No DAPA, se a *thread* t_i processa r regiões, *OpenLists* e *ClosedLists* são divididas em $t * r$ *OpenLists* e $t * r$ *ClosedLists*, onde t é o número total de *threads* e r é o número total de regiões por *thread*. Então, uma *thread* pode processar mais de uma região e o número total de regiões do sistema é $t * r$. A função de *hash* mostrada na Equação 7.1 é então modificada conforme ilustrado na Equação 7.2.

$$(H(C_1, C_2, ..., C_k) \gg S) \bmod (t * r) = t_i \tag{7.2}$$

Como o DAPA irá executar apenas em sistemas onde o uso de memória RAM é alto, podemos assumir que os tamanhos das *OpenLists* e *ClosedLists* são consideráveis. Por esse motivo, foi incluído um novo tipo de lista, chamada *HoldingList*. A *HoldingList* contém nós que foram gerados na iteração atual do algoritmo e deveriam ser inseridos em alguma *OpenList*.

A Figura 7.2 ilustra o projeto do DAPA para 2 *threads*, cada uma com 2 regiões (4 no total). Na primeira iteração do Algoritmo 8, cada nó na fila da *thread* t_i que pertence à região j é adicionado à $OpenList_j$, caso ele pertença a uma região ativa. Caso contrário, ele é adicionado à $HoldingList_j$. É importante observar que um nó somente é adicionado à $OpenList_j$ se ele não está em uma $ClosedList_j$. Se um nó duplicado existir, apenas o nó com maior prioridade é mantido. As *threads* então fazem uma verificação em todas as suas regiões e a região que contém o nó de maior prioridade é marcada como a região ativa. Se a $ClosedList_j$ de um região ativa para esta iteração estiver em disco, ela é transferida de volta à RAM. Após isso, $HoldingList_j$ da região que foi ativada é movida para a $OpenList_j$. Nesta etapa, os nós que estão na $HoldingList_j$ são verificados nas $ClosedList_j$ à busca de duplicatas, novamente mantendo apenas os nós de maior prioridade. Quando a memória RAM utilizada excede o *threshold* , a região que contém o nó de pior prioridade entre todas as regiões é transferida ao disco, até que a quantidade de memória RAM utilizada seja inferior à esse limite. Depois disso, as etapas de *search_ step* e *verify_ end_ condition* do PA-Star são executadas (Algoritmo 8).

A Figura 7.3 representa a leitura e escrita em disco do DAPA. Neste caso, o *threshold* de uso de memória excede um valor definido pelo usuário. Portanto, a *ClosedList* da região 1 é movida para disco. Na figura, a Região 2 está se tornando ativa e, nesse caso, a *ClosedList* é movida de volta à memória.

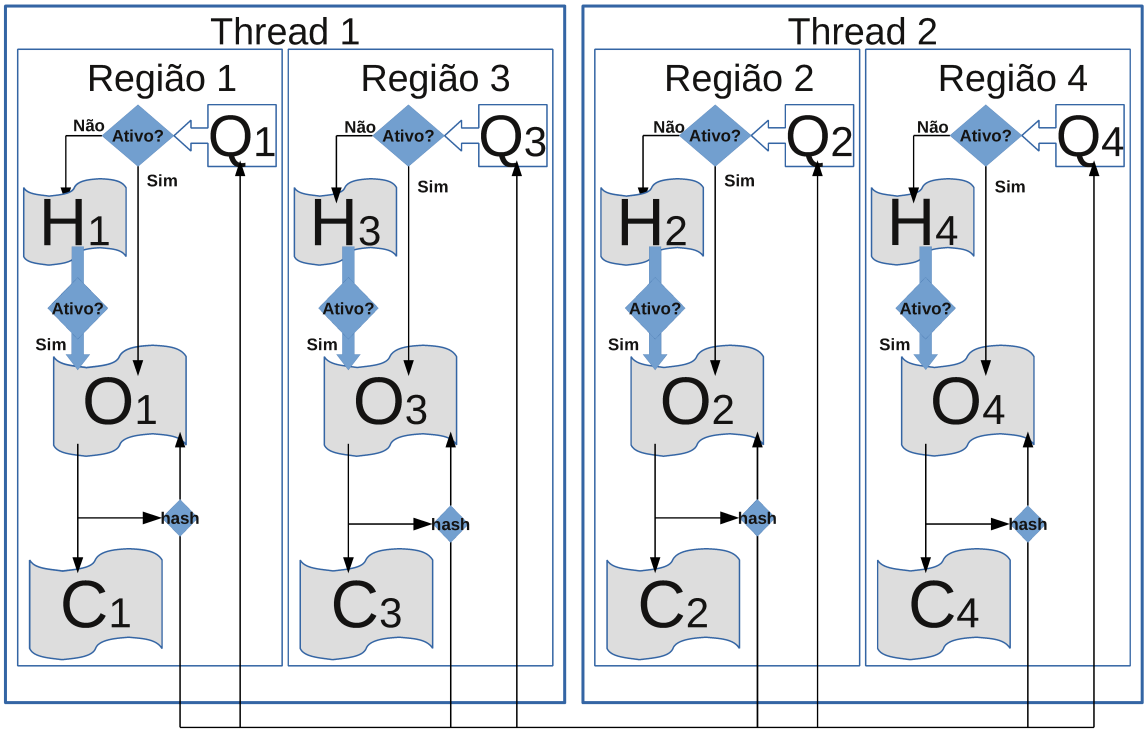


Figura 7.2: Diagrama do DAPA para 2 *threads* e 4 regiões no total, onde a *thread* 1 e 2 são responsáveis pelas regiões 1,3 e 2,4, respectivamente.

7.2 Resultados Experimentais

Nesta seção, apresentamos e discutimos os resultados experimentais obtidos com o PA-Star. A Seção 7.2.1 apresenta o ambiente computacional no qual as sequências foram comparadas. Para analisar a redução de memória da nossa estratégia multi-index, apresentamos na Seção 7.2.2 a comparação entre o PA-Star com estratégia multi-index e o PA-Star com estratégia de dicionário. A Seção 7.2.3 apresenta uma comparação entre as estratégias de alocação dinâmica e *templates*. Um resumo de desempenho de todas as otimizações incluídas no PA-Star é mostrado na Seção 7.2.4. A Seção 7.2.5 compara o desempenho do algoritmo PA-Star utilizando duas funções de *hash* sensível à localidade (Zorder e Sum). A Seção 7.2.6 apresenta o desempenho do PA-Star na comparação de conjuntos de sequências da referência 1 do BAliBASE. A Seção 7.2.7 apresenta resultados que evidenciam a vantagem de usar o disco. Na Seção 7.2.8 comparamos o PA-Star com a ferramenta estado da arte PFA*-DDD. Finalmente, na Seção 7.2.9 nós comparamos o PA-Star, DAPA-PA-Star e PFA*-DDD.

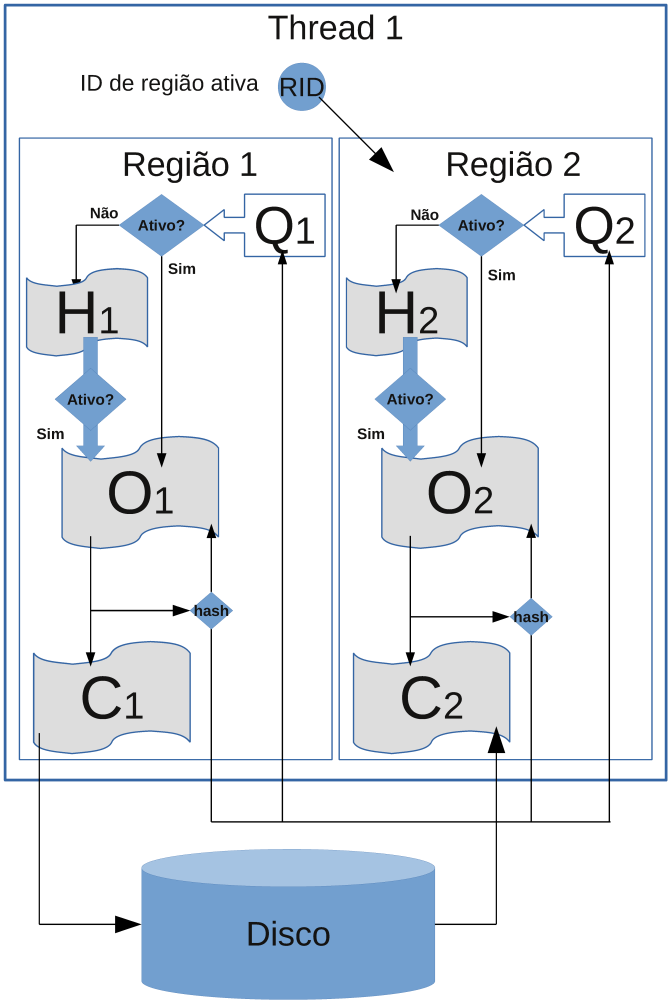


Figura 7.3: Movimento de dados de/para o disco no DAPA quando a Região 2 é marcada como a região ativa e a Região 1 é marcada como inativa. Nós da *ClosedList* da Região 1 são escritos no disco e os nós da *ClosedList* da Região 2 são lidos do disco.

7.2.1 Ambiente Computacional

O PA-Star foi escrito em C++, com *threads* do C++11, com a biblioteca Boost C++ e foi compilado com o compilador g++ com a opção de otimização -O3. Em nossos testes, três máquinas diferentes foram utilizadas, como mostra a Tabela 7.1. Todas as máquina executam Linux CentOS. As duas primeiras máquinas (Máquina1 e Máquina2) são *desktops* disponíveis em nossos laboratórios. A Máquina3 é um servidor localizado na Texas Advanced Computing Center (TACC) acessado através do XSEDE (`http://www.xsede.org`).

Os seguintes parâmetros do PA-Star foram utilizados em nossos testes. A matriz de substituição PAM250 foi utilizada para obter os escores de *matches/mismatches*, ajustados para o problema de minimização do escore (os valores foram adicionados em 17) e a

Tabela 7.1: Máquinas utilizadas nos testes.

Nome	Processador	*Cores*	RAM (GB)	Disco
Máquina1	1 Intel i7-37700 3,5 GHz	4	8	1 TB
Máquina2	1 Intel i7-4790 3,6 GHz	4	32	200 GB
Máquina3	4 Intel Xeon E5-2680 2,7 GHz	32	1.024	250 GB - local 14 PB - Lustre

penalidade de *gap* foi configurada em 30, como em [31]. Em todos os testes, a não ser que especificado, o número de regiões por *thread* foi configurado como 1 e o DAPA (Seção 7.1.4) foi desabilitado.

Os experimentos foram executados utilizando conjuntos de sequências obtidos do *benchmark* BAliBASE [130] disponíveis em `http://labs.bio.unc.edu/Vision/private/Documentation/VisionLab/doc/BAliBASE/align_index.html`. BAliBASE é um banco de dados amplamente utilizado, composto por conjuntos de sequências e os seus alinhamentos múltiplos correspondentes, que são verificados manualmente por biólogos para comparar as ferramentas de alinhamento múltiplo. Em sua versão atual, existem 9 conjuntos de referências. Nos nossos testes, utilizamos todos os 82 conjuntos de sequências que pertencem ao conjunto de referência 1, que contém conjuntos de até 7 sequências de tamanho similar. Também foram utilizadas uma sequência obtida do banco de dados PFAM, disponível em `http://pfam.xfam.org`, além de sequências sintéticas. As sequências synth1 a 3 foram geradas empiricamente, embaralhando alguns blocos de caracteres de forma a favorecer os *gaps* no alinhamento ótimo, como em [124]. A sequência synth4 foi gerada a partir da sequência 2ack do BAliBASE e alguns blocos de caracteres foram embaralhados na sequência, gerando um padrão de alinhamento mais difícil de ser obtido. As características das sequências mencionadas nessa seção são mostradas na Tabela 7.2.

7.2.2 Comparação Multi-index e Dictionary

O objetivo desse experimento é avaliar os ganhos obtidos com o uso da estrutura de dados multi-index. A Máquina2 (Tabela 7.1) foi utilizada nesse teste. Para comparação, duas diferentes abordagens para a implementação da *OpenList* no PA-Star foram consideradas: multi-index e dictionary.

A Tabela 7.3 apresenta o resultado obtido com quatro conjuntos de sequências: PF07708, 3pmg, synth1 e synth2 (Tabela 7.2). O tempo de execução e o uso de memória RAM foram obtidos através do comando *time*. Nesta tabela, podemos notar que, para as sequências comparadas, a abordagem multi-index consome menos memória, e possui um melhor tempo de execução do que a abordagem baseada em dicionário.

Tabela 7.2: Sequências utilizadas nos testes.

Base	Referência	# Seq	Menor	Maior
PFAM	PF07708	22	17	17
BAliBASE	1gdoA	4	235	265
BAliBASE	1dlc	4	568	590
BAliBASE	3pmg	4	540	567
BAliBASE	1wit	5	90	106
BAliBASE	1hva	5	137	199
BAliBASE	arp	5	380	418
BAliBASE	1sesA	5	417	442
BAliBASE	glg	5	438	486
BAliBASE	2ack	5	452	482
BAliBASE	1gpb	5	796	828
BAliBASE	1pamA	5	435	572
BAliBASE	1taq	5	806	928
BAliBASE	1lcf	6	662	691
Sintética	synth1	3	231	416
Sintética	synth2	5	80	98
Sintética	synth3	5	619	619
Sintética	synth4	5	600	600

Neste teste, o melhor resultado foi obtido no conjunto de sequências PF077708, onde o tempo de execução foi reduzido em 28,5% e o uso de memória RAM em 57,7%. Para essa sequência, o uso de memória RAM diminuiu de 28,8 GB para 11,97 GB. A execução do conjunto de sequências synth2 com uma abordagem multi-index obteve uma redução de 0,5% no tempo de execução e 21,4% de redução no consumo de memória, enquanto a execução do synth1 obteve uma redução em 23,2% no tempo de execução e 2,09% no uso de memória. Essas duas sequências foram produzidas manualmente para se obter um conjunto de difícil solução. Mesmo neste caso, multi-index mostrou que é uma melhor abordagem do que dicionário.

7.2.3 Comparação entre *Templates* e Alocação Dinâmica

Neste experimento, utilizamos a Máquina2 (Tabela 7.1) para comparar a implementação do PA-Star utilizando alocação dinâmica de memória e a sua implementação com *templates*. Nessa seção, utilizamos uma *OpenList* multi-index. O PA-Star foi executado com os mesmos conjuntos de sequências utilizados na Seção 7.2.2.

A Tabela 7.4 mostra o tempo de execução e a memória utilizada durante este experimento. Pode ser visto que o tempo de execução e memória são reduzidos consideravelmente quando *templates* são utilizados. Observa-se a redução no tempo de execução

Tabela 7.3: Tempo de execução e uso de memória para as abordagens multi-index e dictionary.

Sequências (k)	Estrutura de Dados	Tempo (s)	RAM (GB)
PF07708 (22)	Dicionário	452,91	28,28
	Multi-index	324,05	11,97
3pmg (4)	Dicionário	75,20	0,73
	Multi-index	66,35	0,59
synth1 (3)	Dicionário	41,14	0,48
	Multi-index	31,58	0,47
synth2 (5)	Dicionário	585,21	3,17
	Multi-index	582,72	2,49

de 18,7% (PF07708) a 43,42% (synth1) e a redução do consumo de memória de 30,58% (PF07708) a 53,20% (synth2) quando se utiliza *templates*. Essa redução pode ser explicada pelo melhor uso de otimizações do compilador, que podem ser aplicadas quando se utiliza *templates*. Neste caso, o compilador sabe exatamente quantas sequências serão utilizadas nas estruturas de dados, permitindo assim melhores otimizações em laços e alocação de memória mais eficiente.

Tabela 7.4: Tempo de execução e uso de memória em abordagens que usam alocação dinâmica de memória e *templates*.

Sequências (k)	Técnica	Tempo (s)	RAM (GB)
PF07708 (22)	Alocação Dinâmica	324,05	11,97
	Templates	263,49	8,31
3pmg (4)	Alocação Dinâmica	66,35	0,59
	Templates	39,97	0,29
synth1 (3)	Alocação Dinâmica	31,58	0,47
	Templates	17,87	0,22
synth2 (5)	Alocação Dinâmica	582,72	2,49
	Templates	358,69	1,22

A Figura 7.4 ilustra a saída do PA-Star para o conjunto de sequências PF07780. Apesar de existirem 22 sequências nesse conjunto e cada nó gerar $2^{22} - 1$ vizinhos durante a fase de expansão, o PA-Star é capaz de obter o alinhamento ótimo em menos de cinco minutos, visto que as sequências são muito similares. Até onde sabemos, é a primeira vez que um alinhamento ótimo múltiplo de sequências foi obtido para um número tão alto de sequências.

```
Starting pairwise alignments... done!
Phase 1 - init heuristic: 00:00.001 s
Performing search with Parallel A-Star.
Running PAStar with: 1 threads, Full-Zorder hash, 12 shift.
Phase 2: PA-Star running time: 04:23.186 s
Final Score: (18 18 18 18 18 18 17 18 17 18 18 17 18 18 18 18 18 18 18 18
18 18)        g - 59579 (h - 0 f - 59579)
Similarity: 59.16%

EEIDPETIKVEVGSDDED
EEIEPEVIRVELGSDEED
EELEPETIPVEIESDEDE
EELDPETIPVDIESDEED
QELDPETTEMELESDEEE
EDLDPETIPVELESDEEE
E-LQPETIPVEVESDDEH
TELEPETIPVELESDDED
E-LEPETIPVEIGSDDEE
EQLKPETIPVEIGSDDEP
EPVEPETIPVEVGSDEEE
G-LQPETIPVEVGSDEDE
QHLQPEHIPVEVGSDDEE
EPLQPERIPVELGSDEEE
QELEPETITVELSSDEEV
EELEPEIFELEISSDSDM
EPLEPETIQVEISSDDED
EHTKPETITVEISSDEEP
EPTEPETITVDLLSSHDE
EPTEPETITVEIESDDDE
EDLDPETIHFEVSSDDEE
FTLQPETIHLEISSDEEE
Phase 3 - backtrace: 00:00.000 s
```

Figura 7.4: Saída produzida pelo PA-Star para o conjunto de sequências PF07780. O PA-Star é capaz de obter o escore ótimo (59.579) e executa um *backtrace* para imprimir o alinhamento ótimo.

7.2.4 Resumo das Otimizações

O impacto causado pelas otimizações propostas, uso da estrutura de dados multi-index com *templates*, além da capacidade de *multithread* é apresentado nas Tabelas 7.5 e 7.6 para os conjuntos de sequências 3pmg e synth2, na Máquina2 (Tabela 7.1), com Hyper-Threading (HT) (Seção 4.1) habilitado.

Na comparação do conjunto 3mpg (Tabela 7.5), o uso de multi-index e *templates* reduziu o tempo de execução em $1,88\times$. Com 8 *threads*, o tempo total é reduzido de $39,97$s para $7,23$s $(5,52\times)$. Com as otimizações e o *multithreading*, o tempo total foi reduzido de $75,20$s para $7,23$s, com um *speedup* total de $10,40\times$ usando as otimizações e 4-HT *cores*.

Tabela 7.5: Impacto das otimizações propostas e *multithreading* para o conjunto 3pmg.

Estrutura de Dados	Técnica	*Threads*	Tempo (s)	RAM (GB)
Dicionário	Aloc. Dinâmica	1	75,20	0,73
Multi-index	Aloc. Dinâmica	1	66,35	0,59
Multi-index	Template	1	39,97	0,29
Multi-index	Template	2	20,03	0,29
Multi-index	Template	4	10,88	0,31
Multi-index	Template	4+4HT	7,23	0,32

Tabela 7.6: Impacto das otimizações propostas e *multithreading* para o conjunto synth2.

Estrutura de Dados	Técnica	*Threads*	Tempo (s)	RAM (GB)
Dicionário	Aloc. Dinâmica	1	585,21	3,17
Multi-index	Aloc. Dinâmica	1	582,72	2,42
Multi-index	Template	1	358,69	1,22
Multi-index	Template	2	191,41	1,24
Multi-index	Template	4	109,48	1,30
Multi-index	Template	4+4HT	74,94	1,40

Na comparação do conjunto synth2 (Tabela 7.6), foi observado o mesmo comportamento, com um *speedup* de $1,63\times$ quando as otimizações de multi-index e *templates* são aplicadas. Com o uso de múltiplas *threads*, o tempo de execução foi reduzido de $358,69$s (1 *thread*) para $74,94$s (8 *threads*), com um *speedup* de $4,78\times$. Com o uso de todas as otimizações e múltiplas *threads* propostas, o PA-Star consegue reduzir o tempo de execução de $585,21$s para $74,94$s, conseguindo uma aceleração de $7,80\times$.

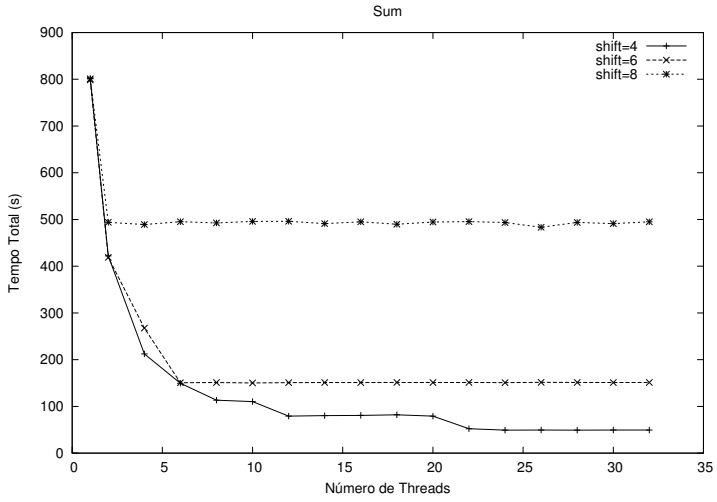
Com as nossas otimizações, também ocorreu uma grande redução no uso de memória utilizada: $2,51\times$ e $2,26\times$ para a execução dos conjuntos 3pmg e synth2, respectivamente. Com o *multithreading*, como esperado, o uso de memória aumenta um pouco devido ao uso de variáveis para sincronização.

7.2.5 Comparação da Função de *Hash*

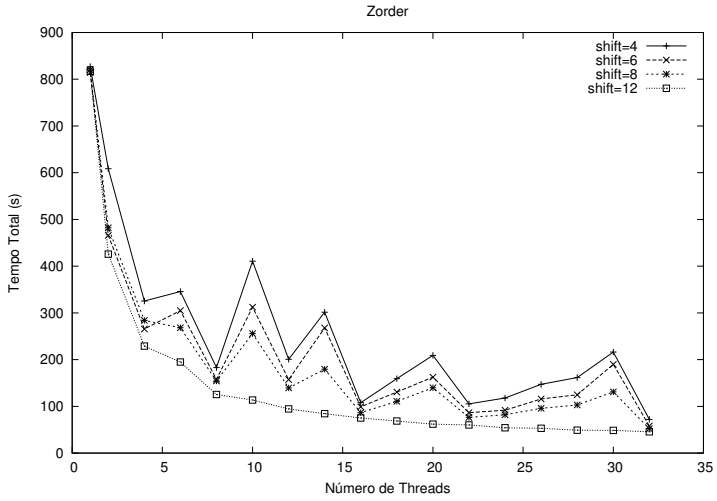
O objetivo desse experimento é avaliar qual função de *hash* deve ser utilizada para distribuir os nós entre as *threads*. Três fatores foram usados para decidir em qual *OpenList* os nós devem ser inseridos (Equações 7.1 e 7.2): (a) a função de *hash* sensível à localidade, Sum ou Zorder; (b) S, uma constante de *shift* à direita; e (c) o número total de *threads* e regiões gerenciadas por cada *thread*.

Se a função de *hash* sempre decidir que o resultado da expansão deve ser inserido em filas que pertencem a outras *threads*, então o *overhead* de comunicação é alto e o

desempenho é afetado. Por outro lado, se a função de *hash* decide que o resultado da expansão nunca deve ser transferido para outras *threads*, então o balanceamento de carga é muito alto e a execução pode ser praticamente serializada. Logo, é importante um compromisso entre esses dois fatores. Neste experimento, utilizamos a Máquina3 (Tabela 7.1) para testar diferentes combinações de funções de *hash*, *threads* e constantes de *shift* à direita.



(a)



(b)

Figura 7.5: Função de *hash* Sum (a) e Zorder (b) para o conjunto de sequências synth2 na Máquina3 (32 *cores*).

A Figura 7.5 mostra o tempo de execução total para a comparação do conjunto de sequências synth2 utilizando as funções de *hash* Zorder (Figura 7.5b) e Sum (Figura 7.5a). Pode ser visto que a função de *hash* utilizada e o fator de *shift* à direita causam um grande impacto no desempenho. No melhor caso, a execução com o *hash* Zorder teve um tempo

total de execução de 45s ($17,9\times$) e o Sum teve de 49s ($16,2\times$). O gráfico também mostra que a Zorder possui resultados mais estáveis para diferentes variações de valores de *shift* à direita, enquanto o desempenho do *hash* Sum rapidamente decaiu para valores mais altos que 4. Em outras palavras, o *hash* Sum parece muito menos escalonável que o Zorder. Em ambos os casos, os números de nós reexpandidos e o *overhead* de comunicação aumenta em relação ao número de *threads*. O gráfico também mostra que há um mínimo local com 4, 8, 16 e 32 *threads*, porque a Equação 7.1 possui um operador *mod* que é otimizado rapidamente como uma operação de *shift*. Com esses resultados, decidimos usar a função de *hash* Zorder por padrão, com um fator de *shift* à direita de 12.

7.2.6 Comparação do BAliBASE

Neste experimento, decidimos executar o PA-Star na Máquina3 (Tabela 7.1) para comparar todas as 82 sequências no referência 1 do BAliBASE, utilizando o *hash* Zorder, 12 de *shift* e 16 *threads*. Foram utilizados 16 *cores* pois empiricamente observamos que a execução para 16 *cores* era ligeiramente melhor que a execução para 32 *cores*.

A maioria dos conjuntos de sequências do BAliBASE referência 1 foi rapidamente resolvida. O tempo total, incluindo o *backtrace*, das 76 instâncias mais fáceis foi de 51 minutos. A execução do PA-Star para três das mais difíceis instâncias do BAliBASE (1sesA, arp e 1gpb) (Tabela 7.2) demorou quase 4 horas para terminar e 3 execuções (1pamA, 1lcf, 1taq) ficaram sem memória na Máquina3, que possui 1TB de memória RAM.

Tabela 7.7: Tempo de execução para 3 das instâncias mais difíceis do BAliBASE 1 (16 *cores*).

Sequências (k)	Comprimento Médio	Tempo (hh:mm:ss)	Tamanho *ClosedList*	Tamanho Lista (Total)
1sesA (5)	427,8	00:30:01	368.609.668	455.761.987
arp (5)	397,0	00:56:43	603.275.186	797.852.355
1gpb (5)	809,8	02:39:27	1.615.316.096	2.009.496.349

A Tabela 7.7 apresenta o tempo de execução para os conjuntos de sequências 1sesA, arp e 1gpb, que são conjuntos de sequências difíceis da referência 1 do BAliBASE. Nesta tabela, pode ser visto que a maior parte dos dados utilizados estão na *ClosedList* (75,61% e 80,87%) e que o número de nós para essa lista é enorme para essas sequências ($>$ 300.000.000).

7.2.7 DAPA

Para avaliar o módulo DAPA, primeiramente fizemos um *profiling* da aplicação para definir o número de *threads* e o número de regiões por *thread*. Escolhemos as sequências $1gdoA$, $1dlc$ e $1wit$ e executamos o PA-Star com DAPA habilitado na Máquina1 (Tabela 7.1). O número de *threads* foi variado de 2 a 8 e o número de regiões por *thread* de 32 a 1.024. Para essa avaliação, o limite de *threshold* foi configurado de forma que o disco não fosse usado. Os resultados para as três sequências foi bastante similar e, por isso, é mostrado o resultado apenas para a sequência $1wit$ na Figura 7.6.

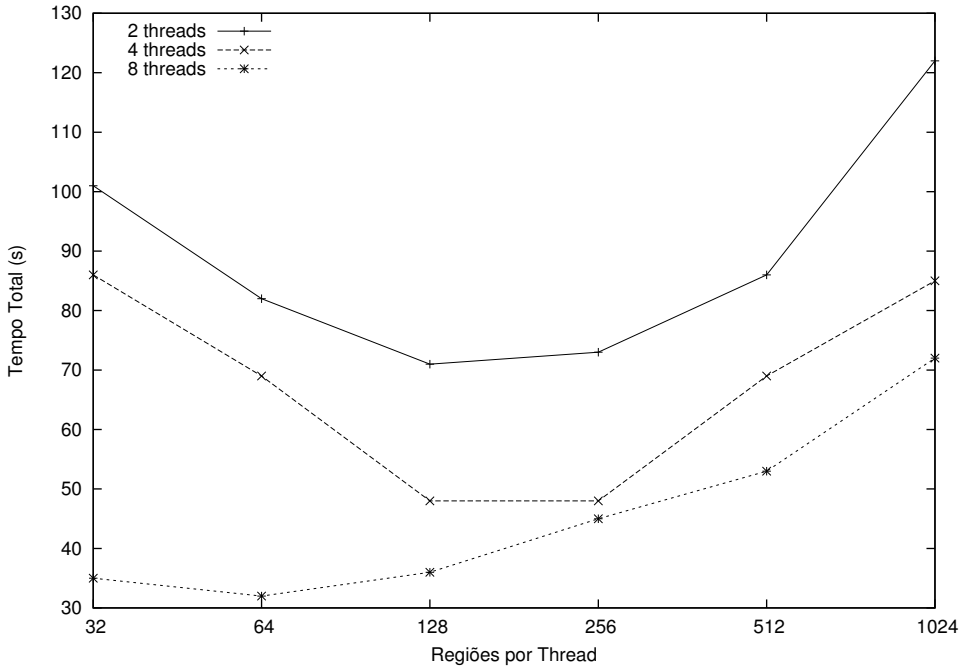


Figura 7.6: Tempo de execução para 2, 4 e 8 *threads* para a comparação do conjunto $1wit$ na Máquina1.

Como é ilustrado na Figura 7.6, o tempo de execução diminui quando o número de regiões é configurado para um certo valor e então ele aumenta novamente. O menor tempo de execução para a Máquina1 foi obtido com 8 *threads* e 64 regiões por *thread*. Portanto, foi escolhida essa configuração para executar o experimento que mede o impacto da memória RAM disponível.

Neste experimento, executamos PA-Star com DAPA habilitado para a sequência $1wit$, utilizando 8 *threads* e 64 regiões por *thread*, com valores decrescente de memória disponível. Em todos os casos, o DAPA iniciou a execução quando 90% do tamanho máximo das *ClosedList* foi atingido. A quantidade de RAM disponível para a *ClosedList* foi variada de 140 MB para 17,5 MB, conforme mostrado na Tabela 7.8. Nesta tabela, o tempo de execução e o uso de memória RAM foram obtidos com o comando *time*. A memória dis-

ponível para a *ClosedList* é um parâmetro do PA-Star, dentro do qual a ocupação máxima da ClosedList foi medida.

Tabela 7.8: Impacto da memória disponível no tempo de execução.

Memória Disponível *ClosedList* (MB)	Tempo de Execução (s)	Ocupação Máxima da *ClosedList* (MB)	Memória RAM (MB)
140,0	15,16	70,59	413,46
70,0	32,77	63,30	399,60
56,0	78,07	51,39	379,29
35,0	436,07	39,94	315,95
17,5	790,58	17,60	272,84

Como esperado, a Tabela 7.8 mostra que a redução da quantidade de memória disponível possui um grande impacto na execução total. Quando toda a *ClosedList* está na memória (140 MB), nenhum acesso a disco é feito e o tempo de execução do PA-Star é $15,16$s. No entanto, quando o disco é necessário, um *overhead* é introduzido e o tempo de execução aumenta.

Na comparação das sequências do $1wit$, quando a memória disponível para a *ClosedList* é reduzida de 140 MB para 17,5 MB ($8\times$), o tempo de execução aumenta de $15,16$s para $790,58$s ($52\times$). Isso é realmente um aumento considerável no tempo de execução, mas nesse caso o DAPA permite que o PA-Star complemente a execução e retorne o resultado ótimo ao usuário.

Quando o PA-Star é executado com o DAPA desabilitado e a quantidade de memória RAM não é suficiente, a execução será finalizada com uma exceção de memória. Para verificar isso, nós utilizamos o conjunto de sequências $1hvA$ (Tabela 7.2) com e sem o DAPA habilitado. Essa comparação exige aproximadamente 10 GB para completar, porém a quantidade disponível de memória RAM na Máquina1 é 8 GB. Portanto, como esperado, a execução do PA-Star sem o DAPA é finalizada por falta de memória.

No teste com o DAPA habilitado, utilizou-se o conjunto de sequência $1hvA$ e inicialmente configurado para 8 *threads* e 64 regiões por *thread*. No entanto, o tempo de execução foi muito alto, então empiricamente aumentamos o número de regiões por *thread* para 256. Com esses parâmetros, PA-Star demorou 7 horas, 14 minutos e 27 segundos para alinhar a sequência $1hvA$. Isso mostra que, com o uso do DAPA, o PA-Star é capaz de encontrar alinhamentos múltiplos que exigem mais memória RAM do que a disponível na estação.

7.2.8 Comparação com o PFA*-DDD

Neste experimento, utilizamos a Máquina3 (Tabela 7.1) para comparar a nossa implementação com outra do estado da arte, a PFA*-DDD (Seção 5.2.4). Para fazer essa comparação, os autores do PFA*-DDD nos forneceram o código fonte. O PFA*-DDD exige uma biblioteca MPI e neste teste utilizamos a Intel IMPI versão 5.0.1.035. O PFA*-DDD exige também processos múltiplos de 3 para executar, enquanto a nossa solução pode ser executada com qualquer número de *threads*.

A Figura 7.7 mostra o tempo total de execução para a sequência 2ack e glg (Tabela 7.2). Neste teste, as sequências foram cortadas para que tivessem tamanho uniforme (450 e 438, respectivamente). Fica evidente pela Figura 7.7 e Tabela 7.9 que o PA-Star sempre possui um tempo de execução melhor, sendo o melhor resultado obtido com 32 *threads*. O PFA*-DDD minimiza o seu tempo de execução quando utiliza 48 processos.

Tabela 7.9: Comparação entre a nossa estratégia paralela do A-Star (com 32 *threads*) e o PFA*-DDD (com 48 processos).

Sequência	PA-Star	PFA*-DDD
	Tempo (s)	Tempo (s)
2ack	62,64	299,21
glg	30,76	89,15

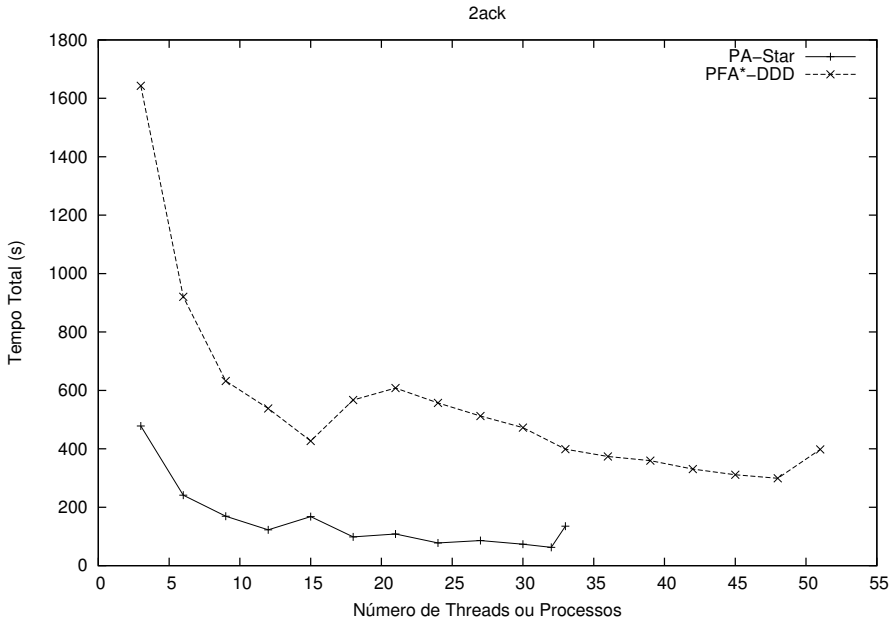
A Tabela 7.9 mostra o melhor tempo de execução para ambas as soluções. Como pode ser visto, PA-Star compara a sequência 2*ack* em 1 minuto e 2 segundos, enquanto o PFA*-DDD demora mais de 5 minutos para fazer a mesma comparação. Para a sequência *glg*, o tempo de execução do PA-Star foi de 30,76 segundos, enquanto o PFA*-DDD executa em um tempo total de 89,15 segundos.

7.2.9 Comparação do PA-Star, PA-Star-DAPA e PFA*-DDD

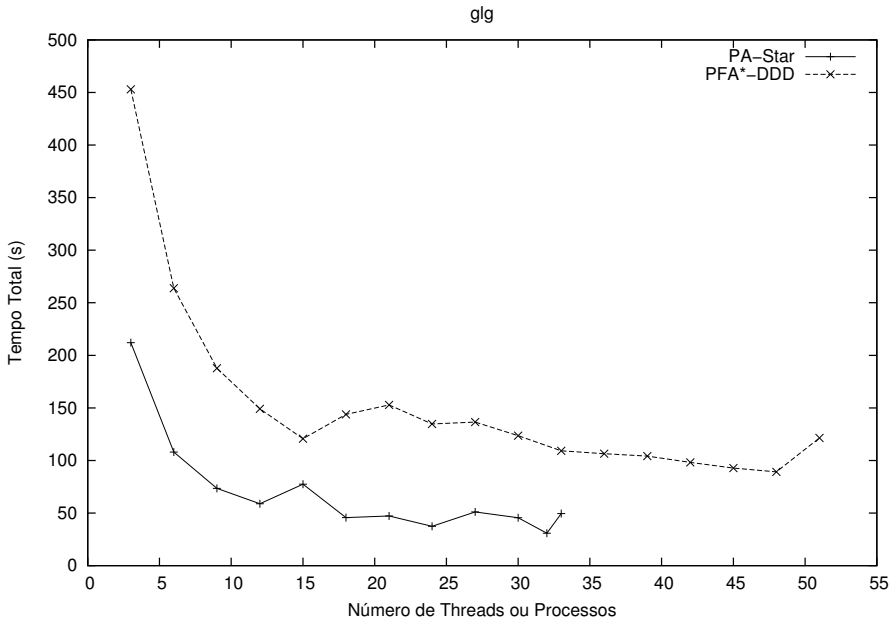
Neste experimento, comparamos o PFA*-DDD (Seção 5.2.4) com o PA-Star sem DAPA e o PA-Star com DAPA. Foi utilizada a Máquina1, executando PFA*-DDD com o Intel IMPI versão 5.0.1.

Tabela 7.10: Comparação entre a nossa estratégia Paralela PA-Star (com 8 *threads*), DAPA (com 4 *threads* e 256 regiões) e PFA*-DDD (com 12 processos).

Sequência (k)	PA-Star-DAPA (s)	PA-Star (s)	PFA*-DDD (s)
synth4 (5)	5.604	sem memória	sem memória



(a)



(b)

Figura 7.7: Comparação entre PA-Star e PFA*-DDD na Máquina3 (32 *cores*).

A Tabela 7.10 mostra o tempo de execução em segundos do PA-Star com DAPA, PA-Star sem DAPA e PFA*-DDD, quando estão executando o conjunto de sequências synth4 (Tabela 7.2). Como pode ser visto, o PA-Star com DAPA é capaz de calcular o alinhamento ótimo para a *synth*4 em 1 hora e 33 minutos, enquanto o PFA*-DDD e PA-Star não terminaram a execução devido à memória RAM insuficiente.

Neste experimento, o PA-Star foi criado com 8 *threads* e fica sem memória em 8

minutos. O PFA*-DDD criou 12 processos e terminou a execução por falta de memória depois de 42 minutos e 49 segundos. O PA-Star com DAPA criou 4 *threads* e 256 regiões por *thread*, com uma quantidade disponível de RAM para *ClosedList* configurada para 1,5 GB. Esse teste mostra que o DAPA é capaz de calcular o alinhamento múltiplo ótimo que exige uma quantidade maior de RAM que a disponível na máquina, enquanto as outras duas soluções terminam a execução com uma exceção de memória.

7.2.10 Comparação com o Estratégias de Outros Domínios

Tabela 7.11: Comparação entre estratégias paralelas baseadas em A-Star.

Nome	Estratégia Paralela	Uso de Disco	Sensível à Localidade	Domínio
PFA*-DDD [89]	MPI	Não	Não	MSA
PE2A* [37]	*Multithread*	**Sim**	Não	MSA
GA* [140]	GPU	Não	Não	SPu, PF, PDe
PEMM [120]	*Multithread*	**Sim**	Não	RC
AZHDA* [59]	*Multithread*	Não	**Sim**	SPu, MSA
PA-Star	*Multithread*	**Sim**	**Sim**	MSA

Na literatura são encontrados diversas soluções paralelas para o algoritmo A-Star (Seção 5.5). A Tabela 7.11 mostra a comparação dessas estratégias com o PA-Star, mostrando a estratégia paralela utilizada, o uso de disco, o uso de funções de *hash* sensíveis à localidade e o problema que o A-Star foi aplicado: Alinhamento Múltiplo de Sequências (MSA), Pathfinding (PF), Sliding Puzzle (SPu), Protein Design (PDe) e Rubik Cube (RC). Esta tabela mostra que o PA-Star avançou o estado da arte, sendo a única solução que, até onde sabemos, utiliza função sensível à localidade para a distribuição de nós e disco.

7.3 Conclusão do Capítulo

Nesta capítulo propusemos e avaliamos o PA-Star, uma solução paralela capaz de obter o alinhamento múltiplo ótimo em múltiplos *cores*. Foram propostas muitas otimizações, tais como uso da estrutura multi-index, *templates* e uma função de *hash* sensível à localidade e disco, quando a memória RAM não é suficiente. Para o PA-Star, foi também proposta uma estratégia para diminuir o número de sincronizações entre as *threads*, acelerando a execução.

Os resultados obtidos com conjuntos de sequências do BAliBASE, referência 1, mostram que o PA-Star é capaz de reduzir o tempo de execução total de $75,20$s para $7,23$s, adicionando a estrutura multi-index, usando *templates* e executando paralelamente em 8

cores. Usando o PA-Star, nós fomos capazes de comparar 79 conjuntos de sequência da referência 1 do BAliBASE em 4 horas e 56 minutos, produzindo o alinhamento ótimo para todos esses conjuntos. Foi também mostrado que o PA-Star possui um melhor desempenho do que outra ferramenta estado da arte, o PFA*-DDD (Seção 5.2.4), executando até $4,77\times$ mais rápido. Finalmente, foi mostrado que nosso módulo de disco DAPA é capaz de calcular o alinhamento ótimo enquanto o PFA*-DDD e PA-Star sem o módulo de disco não finalizam a execução por falta de memória.

Capítulo 8

Foldalign 2.5: Estratégia *multithreaded* para o Alinhamento Estrutural Heurístico em Pares de Sequências de RNA

O Foldalign 2.5 [123] é a segunda contribuição desta tese. Conforme explicado na Seção 3.3.5.1, o Foldalign utiliza heurísticas para reduzir a alta complexidade computacional do algoritmo de Sankoff (Seção 3.3.2). Mesmo com essas heurísticas, uma única execução pode demorar várias horas. Neste capítulo, propomos uma estratégia que utiliza várias *threads* para explorar o padrão de *wavefront* da matriz de programação dinâmica do Foldalign, reduzindo o tempo total de execução para obter o alinhamento estrutural em pares de sequências de RNA.

O objetivo deste capítulo é apresentar o Foldalign 2.5. O projeto da estratégia e o algoritmo paralelo são explicados na Seção 8.1. O ambiente computacional, as sequências utilizadas e os resultados obtidos são descritos na Seção 8.2. Finalmente, uma discussão sobre a nova versão do Foldalign é apresentada na Seção 8.3.

8.1 Projeto do Foldalign 2.5

São encontrados dois desafios na implementação de uma estratégia *multithread* para o Foldalign. Primeiro, é necessário o cálculo da matriz de programação dinâmica 4D entre as *threads*. Explorar esse paralelismo não é trivial, afinal o algoritmo define uma matriz de programação dinâmica 4D de tamanho não uniforme e com uma complexa dependência de dados. Segundo, deseja-se manter todas as funcionalidades de versões anteriores do Foldalign, dentre elas a heurística de *pruning* (Seção 3.3.5.1), que descarta

dinamicamente células do espaço de busca de acordo com seu escore e uma constante informada pelo usuário. Desta forma, não se sabe a *priori* quantas células da matriz de programação dinâmica cada *thread* irá processar. Por esse motivo, é necessário um mecanismo adicional de sincronização entre as *threads* pois o tempo de execução delas pode ser bem diferente.

8.1.1 Algoritmo Paralelo Proposto

O algoritmo do Foldalign 2.5 está ilustrado no Algoritmo 10. Ele recebe como parâmetro duas sequências A e B de tamanho L_A e L_B, respectivamente. Nele, existem seis *loops* "for" aninhados (linhas 1 a 4, 18 e 19), sendo que o primeiro é executado em paralelo por um número de *threads* determinado pelo usuário. O primeiro e segundo *loops* (linhas 1 e 2) processam as sequências de trás pra frente (valores $i = L_A, L_A{-}1, \ldots, 1$ e $k = L_B, L_B{-}1, \ldots 1$). Os *loops* das linhas 3, 4, 18 e 19 usam as heurísticas λ e δ para reduzir a complexidade de tempo e memória do algoritmo.

O Foldalign 2.5 divide a matriz de programação dinâmica em *long term memory* (LTM), para células que podem fazer parte de uma regra de *multibranch* e são mantidas em memória por um período maior de tempo e *short term memory* (STM), para as outras células. No algoritmo, elas são representadas pelas matrizes L e S, respectivamente. Na linha 6, células que podem ser parte de uma regra de *multibranch* são copiadas da STM para a LTM. Nas linhas 9 a 17 são aplicados os termos da equação de recorrência para *matches*, *mismatches*, *gaps*, pares de bases e nucleotídeos não pareados. Das linhas 18 a 22 é aplicada a regra de *multibranch*, para combinar duas estruturas menores em uma maior.

O Foldalign serial utiliza uma abordagem baseada em colunas, onde uma coluna na matriz de programação dinâmica é totalmente processada antes de iniciar o processamento da próxima coluna. O modelo *multithreading* do Foldalign 2.5 é baseado no fato que alguma células de colunas adjacentes podem ser calculadas em paralelo, com um padrão de *wavefront* nas anti-diagonais, sem ser necessário o processamento completo da coluna anterior.

A Figura 8.1 ilustra as diferenças entre a estratégia serial e a estratégia paralela proposta. Na versão serial, a *thread* calcula completamente uma coluna antes de iniciar o processamento da próxima. Na versão paralela, são criadas $t = 1, 2, ..., N$ *threads*, e cada uma delas processa uma coluna de valor $i_t = i, i-1, ..., i-N$. Assim, cada *thread* calcula sequencialmente as células $(i_t, L_B) \rightarrow (i_t, 1)$. Quando uma *thread* termina de calcular todas as células i_t, ela reinicia com um novo valor $i_t = i_t - N$.

Na Figura 8.1 é mostrada a matriz de programação dinâmica 4D, com os eixos i e k. Cada célula dessa matriz é uma outra matriz bidimensional, de coordenadas W_i e W_k. Na figura, as células em vermelho e azul escuro são as células que já foram processadas pelas

Algoritmo 10 :`Foldalign 2.5 DP(A, B)`

1: **for** $i = L_A \rightarrow 1$ **do** {For paralelo}
2: **for** $k = L_B \rightarrow 1$ **do**
3: **for** $W_i = 1 \rightarrow \lambda$ **do**
4: **for** $W_k = (W_i - \delta) \rightarrow (W_i + \delta)$ **do**
5: $s = S_{i,k,W_i,W_k}$
6: **if** $pode\_ser\_multibranch(S_{i,k,W_i,W_k})$ **then**
7: $L_{i,k,W_i,W_k} = s$
8: **end if**
9: $S_{i-1,k-1,W_{i+2},W_{k+2}} = \max(s + E_{bp}(i-1, k-1, W_{i+2}, W_{k+2}),$
$S_{i-1,k-1,W_{i+2},W_{k+2}})$
10: $S_{i,k-1,W_i,W_{k+2}} = \max(s + E_{bpiI}(i, k-1, W_i, W_{k+2}), S_{i,k-1,W_i,W_{k+2}})$
11: $S_{i-1,k,W_{i+2},W_k} = \max(s + E_{bpiK}(i-1, k, W_{i+2}, W_k), S_{i-1,k,W_{i+2},W_k})$
12: $S_{i-1,k-1,W_{i+1},W_{k+1}} = \max(s + E_{al}(i-1, k-1, W_{i+1}, W_{k+1}), S_{i-1,k-1,W_{i+1},W_{k+1}})$
13: $S_{i,k,W_{i+1},W_{k+1}} = \max(s + E_{ar}(i, k, W_{i+1}, W_{k+1}), S_{i,k,W_{i+1},W_{k+1}})$
14: $S_{i,k-1,W_i,W_{k+1}} = \max(s + E_{glI}(i, k-1, W_i, W_{k+1}), S_{i,k-1,W_i,W_{k+1}})$
15: $S_{i-1,k,W_{i+1},W_k} = \max(s + E_{grI}(i-1, k, W_{i+1}, W_k), S_{i-1,k,W_{i+1},W_k})$
16: $S_{i,k,W_i,W_{k+1}} = \max(s + E_{glK}(i, k, W_i, W_{k+1}), S_{i,k,W_i,W_{k+1}})$
17: $S_{i,k,W_{i+1},W_k} = \max(s + E_{grK}(i, k, W_{i+1}, W_k), S_{i,k,W_{i+1},W_k})$
18: **for** $W_m = 1 \rightarrow (\lambda - W_i)$ **do**
19: **for** $W_n = (W_m - \delta) \rightarrow (W_m + \delta)$ **do**
20: $S_{i,k,W_i+W_m,W_k+W_n} = \max(s + L_{i+W_i+1,k+W_k+1,W_m,W_n} +$
$E_{i,k,W_i+Wm,W_k+W_n}, S_{i,k,W_i+W_m,W_k+W_n})$
21: **end for**
22: **end for**
23: **end for**
24: **end for**
25: **end for**
26: **end for**

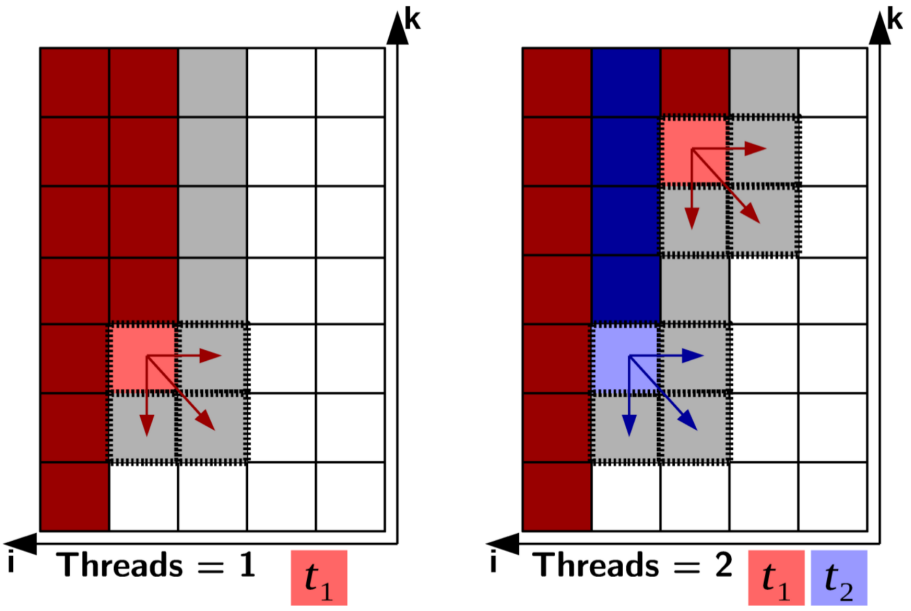


Figura 8.1: Exemplo de execução de processamento da matriz de programação dinâmica do Foldalign 2.5 com uma *thread* (esquerda) e duas *threads* (direita).

threads t_1 e t_2, respectivamente. As células em vermelho e azul claro são as células em processamento pelas *threads* e as células em cinza e branco são as células que ainda serão processadas. A região pontilhada representa as células que estão sendo lidas e escritas pelas *threads*.

8.1.2 Sincronização entre *Threads*

A matriz de programação dinâmica do Foldalign possui um padrão de crescimento não regular. Assim, algumas *threads* podem processar suas regiões da matriz de programação dinâmica mais rapidamente do que outras, principalmente porque algumas células podem ser descartadas pela heurística do *pruning*. Por causa disso, é necessário um mecanismo de sincronização entre as *threads* para garantir que não irão ocorrer condições de corrida e que uma célula seja processada somente depois de todas as outras células das quais depende.

Para isso, devemos sincronizar as *threads*. De acordo com a dependência de dados da equação de recorrência do Foldalign, uma célula (i, k) escreve nas posições $(i-1, k)$, $(i, k-1)$ e $(i-1, k-1)$. Porém, a célula $(i+1, k-1)$ também escreve na coordenada $(i, k-1)$. Para evitar escritas simultâneas na mesma região da matriz dinâmica, a célula (i, k) deve ser processada apenas depois que a célula de coordenadas $(i+1, k-1)$ já foi processada.

Na estratégia paralela proposta, reservamos em uma memória global um *array* contendo as variáveis i e k de cada uma das N *threads*, além de adicionar variáveis de condição e uma variável booleana, inicializada como verdadeiro, para dizer se a *thread* está ativa ou não. Essas variáveis estão disponíveis para leitura por todas as outras *threads*, porém somente a *thread* N pode alterar as variáveis do N-ésimo elemento deste *array*.

Quando uma *thread* t_i inicia a iteração k de um *loop* (Algoritmo 10 linha 2), ela verifica se a *thread* t_{i-1} (ou t_N, se $i = 1$) processou a posição $(i{+}1, k{-}1)$ lendo os valores dos contadores disponíveis na memória global. Se a célula não foi processada, ela altera a variável boolena para falso e a *thread* t_i aguarda na sua variável de condição. Além disso, quando uma *thread* t_i termina de processar a uma célula (i, k), ela verifica se a *thread* t_{i+1} (ou t_1, se $i = N$) está ativa. Caso contrário, ela sinaliza a variável de condição da *thread* t_{i+1}, avisando que pode continuar o processamento.

Esse mecanismo protege as células que estão sendo escritas na STM, porém também permite acessos concorrentes à LTM. Porém, o acesso à essa memória não é regular e diversas *threads* podem acessar concorrentemente diversas posições. Para a solução desse problema, é adicionada uma proteção por *mutex* para cada posição (i, k) da matriz LTM. Sendo assim, antes de realizar operações de leitura e escrita na matriz LTM, cada *thread* obtém o *mutex* e o libera ao final de seu acesso. Empiricamente, observamos que esse *overhead* de comunicação não é considerável e, na maior parte das vezes, as *threads* estão acessando diferentes posições (i, k) desta matriz.

8.1.3 Estratégia de Criação de *Threads*

A abordagem paralela utilizada no Foldalign 2.5 é mestre-trabalhador. A primeira *thread*, criada no início da execução do programa, é a *thread* mestre. Ela é responsável por criar as outras *threads* e distribuir o trabalho entre as *threads* trabalhadoras.

Foram implementadas duas estratégias de criação de *threads*: dinâmica e estática. Na estratégia dinâmica, uma *thread* trabalhadora é criada no início do processamento de uma coluna k da matriz de programação dinâmica. Ao término do processamento desta coluna, a *thread* trabalhadora é destruída. Na estratégia estática, um número fixo de *threads* trabalhadoras é criado no início do programa, essas *threads* aguardam o trabalho em um *pool* de *threads*. Quando uma *thread* termina o processamento de uma coluna k, ela avisa a *thread* mestre o término de seu trabalho e volta a esperar trabalho em um *pool de threads*.

8.2 Resultados Experimentais

Nesta seção, apresentamos e discutimos os resultados experimentais obtidos com o Foldalign 2.5. A Seção 8.2.1 apresenta o ambiente computacional dos testes, assim como as sequências que foram utilizadas. A Seção 8.2.2 compara duas abordagens de criação de *threads* propostas. A Seção 8.2.3 mostra os *speedups* obtidos para diferentes sequências. Uma comparação do tempo de execução e uso de diferentes valores da heurística δ é mostrada na Seção 8.2.4. Uma análise do tempo de execução e *speedups* para sequências com diferentes *GC-Content* (Seção 3.4.1) é apresentado na Seção 8.2.5. Finalmente, um exemplo de uso do *Webserver* atualizado do Foldalign 2.5 é apresentado na Seção 8.2.6.

8.2.1 Ambiente Computacional dos Testes e Sequências Utilizadas

O Foldalign 2.5 foi escrito em C++ utilizando POSIX *Threads* (Pthreads) (Seção 4.1.1) para a manipulação de *threads*. Os mecanismos de sincronização utilizados foram variáveis de condição ($pthread\_cond\_t$) e o *mutex* ($pthread\_mutex\_t$). O Foldalign foi compilado com o compilador g++ com a opção de otimização -O3.

Os experimentos foram executados em uma máquina com 2 processadores Intel Xeon E5-2650, cada um com 8 *cores*, 2,00 GHz e 32 GB de RAM. Em todos os testes, a não ser quando especificado o contrário, são utilizados os parâmetros padrão do Foldalign: execução com 1 *thread*, $\lambda = 1.000$, $\delta = 25$ e a heurística de *pruning* habilitada.

Tabela 8.1: Sequências sintéticas e reais utilizadas nos experimentos.

Base	Referência	# Seq	Tamanho Médio
Sintética	Synth1	2	1.000
Sintética	Synth2	2	2.000
Sintética	Synth3	2	3.000
Real	Real1	2	2.167
Sintética	Synth4	2	9.380

Nos experimentos das seções 8.2.2 a 8.2.4 e 8.2.6, utilizamos sequências sintéticas e reais (Tabela 8.1). Essa tabela contém 5 sequências, sendo 1 real e 4 sintéticas. As sequências *Synth*1, *Synth*2 e *Synth*3 são sequências geradas aleatoriamente, com o objetivo de medir a performance do Foldalign. A sequência *Real*1 contém um par de sequências de RNA 16S Ribossomal obtidas no endereço `http://www.rna.ccbb.utexas.edu/` [7].

A sequência *Synth*4 foi gerada a partir de 2 sequências da família RF00167 do banco de dados RFAM (Seção 3.4.2). Essas sequências contém um *Purine riboswitch* e não possuem alta similaridade na sequência primária. Originalmente, essas sequências possuem

650 e 690 nucleotídeos. Para gerar as sequências sintéticas, esses *riboswitches* foram concatenados algumas vezes para gerar uma sequência intermediária. Então, essa sequência intermediária foi novamente concatenada algumas vezes e algumas dessas concatenações foram embaralhadas, resultando em sequências de tamanho 9.100 e 9.660 nucleotídeos, que possuem um estrutura secundária estável no meio delas.

8.2.2 Comparação entre *Threads* Dinâmicas e Estáticas

O objetivo desse experimento é avaliar a diferença de desempenho entre as estratégias de criação de *threads* dinâmica e estática no Foldalign 2.5. Para isso, utilizamos as sequências *Synth*1 para avaliar o desempenho do Foldalign 2.5 para 1 a 8 *threads* trabalhadoras.

Tabela 8.2: Tempo de execução de n *threads*, utilizando o conjunto *Synth*1 utilizando criação de *threads* estáticas e dinâmicas.

Número de threads	Estática (s)	Dinâmica (s)	Redução no tempo
1	35,95	36,18	0,64%
2	29,70	29,81	0,37%
3	20,33	24,79	17,99%
4	19,57	22,38	12,56%
5	18,78	21,48	12,57%
6	17,89	20,96	14,65%
7	17,86	19,76	9,62%
8	17,96	18,77	4,32%

Como pode ser visto na Tabela 8.2, a estratégia estática possui um melhor desempenho que a estratégia dinâmica. Para uma abordagem que utiliza apenas uma *thread* trabalhadora, a diferença de desempenho entre as estratégias é de apenas 0,64%, mas ao utilizar mais *threads*, o ganho chega a 17,99% no caso de 3 *threads*. Isso pode ser explicado pela estratégia estática possuir um menor *overhead* de criação e término de *threads*.

Entre 3 *threads* e 8 *threads*, o ganho de desempenho é reduzido a cada *thread* adicionada, chegando a 4,32% para 8 *threads*. Para o conjunto de sequências *Synth*1, não existe ganho de desempenho ao utilizar 6 ou 8 *threads* na estratégia estática. Com isso, acreditamos que o paralelismo já esteja no máximo, tornando o overhead de criação de *threads* com menor impacto no tempo de execução total.

Com base nesses resultados, decidimos utilizar a estratégia estática como padrão no Foldalign 2.5 e em todos os outros experimentos descritos neste capítulo.

8.2.3 Tempo de Execução e *Speedup* e Consumo de Memória

Neste experimento foi avaliada a redução do tempo de execução total e *speedup* utilizando um conjunto de sequências longas com 6.000 nucleotídeos (*Synth*2). Esse conjunto possui 5 pares de sequências, ambos com *GC-Content* (Seção 3.4.1) entre 40% e 50%. Para este experimento, calculamos cada par separadamente e calculamos o tempo e consumo de memória médio.

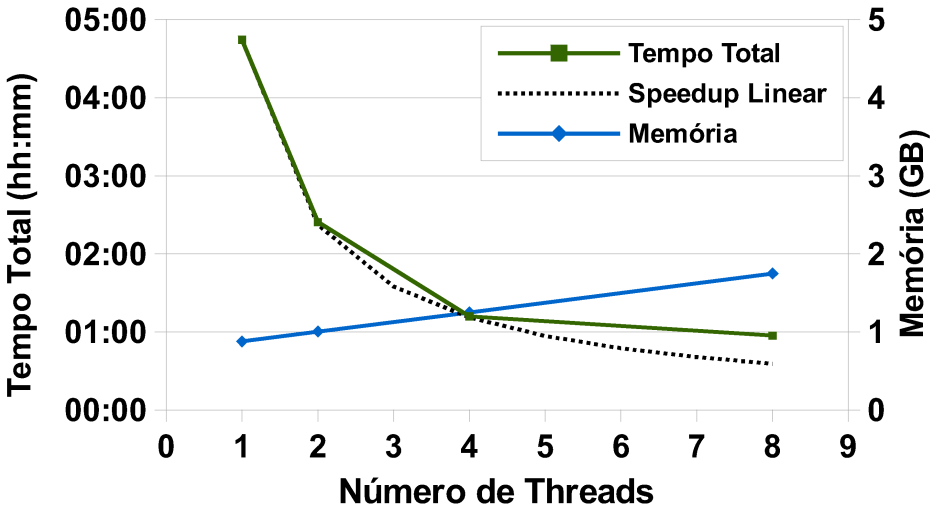


Figura 8.2: Tempo de execução e consumo de memória de acordo com o número de *threads*.

A Figura 8.2 ilustra os resultados. Nela, a linha verde representa o tempo total de execução, a linha azul representa o consumo de memória e a linha pontilhada representa o *speedup* linear, onde n *threads* são utilizadas para obter a execução n vezes mais rápido.

Usando 8 *threads* o tempo total de execução é reduzido de 4 horas e 44 minutos para 57,1 minutos, sendo $4,98$ vezes mais rápido. Utilizando 2 ou 4 *threads*, a redução é, respectivamente, para 2 horas e 24 minutos ($1,97$ vezes mais rápido) e 1 hora e 11 minutos ($3,95$ vezes mais rápido).

Também podemos ver pela Figura 8.2 que o consumo de memória não aumenta consideravelmente com o número de *threads*. Ao utilizar 2 *threads*, o consumo de memória aumenta $1,14$ vezes e ao utilizar 8 *threads*, o consumo aumenta $1,99$ vezes. Esse aumento pode ser explicado pois mais colunas k da memória STM são mantidas em memória.

8.2.4 Impacto da Heurística δ no Tempo de Execução

O objetivo desse experimento é obter a redução de tempo utilizando sequências sintéticas *Synth*3 (Tabela 8.1) e sequências reais *Real*1 (Tabela 8.1). Para cada um desses pares de sequências, diferentes valores de δ foram utilizados.

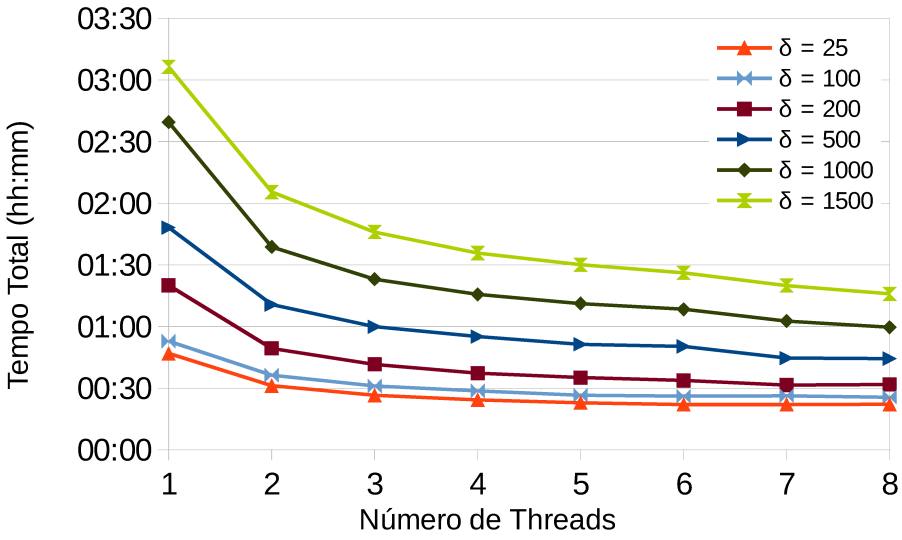


Figura 8.3: Alinhamento de duas sequências aleatórias (*Synth*3) de tamanho 3.000 para diferentes *threads* e valores de δ.

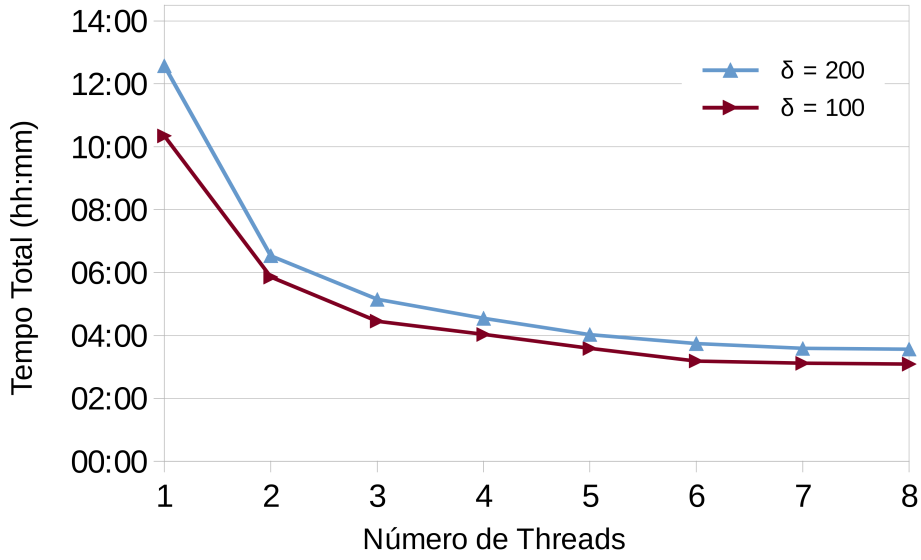


Figura 8.4: Tempo total de execução de duas sequências reais (*Real*1) para diferentes *threads* e valores δ.

A Figura 8.3 mostra o tempo de execução para o par de sequências *Synth*3. Pode-se observar na figura que a heurística δ possui um fator considerável de redução de tempo, reduzindo o tempo de execução de mais de 3 horas ($\delta = 1.500$) para 47 minutos ($\delta = 25$). Porém, em casos onde o δ é muito baixo, o *speedup* é desfavorecido e o tempo de execução se mantém praticamente estável a partir de 3 *threads*, mostrando que para os valores $\delta = 25$ e 100 a execução é rápida, reduzindo o potencial para explorar o paralelismo.

A Figura 8.4 mostra o tempo de execução para o par de sequências *Real*1. O tempo de execução do Foldalign 2.5 para o conjunto *Real*1 foi consideravelmente maior que o conjunto *Synth*3 e por isso os resultados foram obtidos apenas para os valores $\delta = 100$ e 200. Pode-se notar que, para esse conjunto, o uso de δ mais alto também introduz um tempo de execução maior nas sequências, porém os *speedups* obtidos para δ mais alto são melhores.

Além disso, comparando as duas figuras, é possível ver que o tempo de execução do par de sequências aleatórias (Figura 8.3) é consideravelmente menor que do que o do par de sequência reais (Figura 8.4). Isso acontece pois sequências aleatórias geralmente não possuem uma estrutura estável. O alinhamento da Figura 8.4 possui 808 nucleotídeos, enquanto o alinhamento da Figura 8.3 possui apenas 17 nucleotídeos. Isso pode ser explicado pela heurística de *pruning*, que descarta mais células do espaço de busca. Desta forma, um alinhamento de sequências com tamanho médio de 3.000 nucleotídeos pôde se executar mais rápido do que o alinhamento de sequências de tamanho 2.167. Também observamos que valores mais altos de δ aumentam o tempo total de execução, mas favorecem os *speedups* na proposta paralela.

8.2.5 Impacto do *GC-Content* no Tempo de Execução e Consumo de Memória

O objetivo desse experimento é avaliar o impacto do *GC-content* das sequências no tempo total de execução e *speedup* obtido com o algoritmo *multithreaded* proposto. Para isso, utilizamos os conjuntos de sequências da Tabela 8.3. Esses conjuntos possuem sequências de tamanhos de 2.000, 3.000 e 6.000 nucleotídeos e foram organizados de forma a combinar diferentes valores de *GC-Content*: de 20% a 30%, de 30% a 40%, de 40% a 50% e de 50% a 60%. Os conjuntos também possuem a mesma proporção de nucleotídeos G/C e A/U. e são formados por 5 pares de sequências aleatórias. Neste experimento, calculamos o valor médio de consumo de memória e o tempo total de execução.

As Tabelas 8.4 a 8.5 mostram o tempo de execução e consumo de memória, para as sequências de tamanho 2.000 e 3.000, utilizando a opção de $\lambda = 1.000$. A Tabela 8.6 mostra o tempo de execução e consumo de memória para os conjuntos de tamanho 6.000 ($\lambda = 1.000$).

Para conjuntos com as sequências de tamanho 2.000 (Tabela 8.4) e tamanho 3.000 (Tabelas 8.5) é possível observar uma redução no consumo de memória para valores menores de λ. Esse resultado é esperado, afinal essa heurística reduz a complexidade de memória do algoritmo.

Tabela 8.3: Conjuntos de sequências aleatórias com diferentes valores de *GC-Content* utilizados nos experimentos.

Ref.	*GC-Content* Sequência 1	*GC-Content* Sequência 2	Tamanho	Ref.	*GC-Content* Sequência 1	*GC-Content* Sequência 2	Tamanho
GC01	20%-30%	20%-30%	2.000	GC11	20%-30%	20%-30%	3.000
GC02	20%-30%	30%-40%	2.000	GC12	20%-30%	30%-40%	3.000
GC03	20%-30%	40%-50%	2.000	GC13	20%-30%	40%-50%	3.000
GC04	20%-30%	50%-60%	2.000	GC14	20%-30%	50%-60%	3.000
GC05	30%-40%	30%-40%	2.000	GC15	30%-40%	30%-40%	3.000
GC06	30%-40%	40%-50%	2.000	GC16	30%-40%	40%-50%	3.000
GC07	30%-40%	50%-60%	2.000	GC17	30%-40%	50%-60%	3.000
GC08	40%-50%	40%-50%	2.000	GC18	40%-50%	40%-50%	3.000
GC09	40%-50%	50%-60%	2.000	GC19	40%-50%	50%-60%	3.000
GC10	50%-60%	50%-60%	2.000	GC20	50%-60%	50%-60%	3.000

Ref.	*GC-Content* Sequência 1	*GC-Content* Sequência 2	Tamanho
GC21	20%-30%	20%-30%	6.000
GC22	20%-30%	30%-40%	6.000
GC23	20%-30%	40%-50%	6.000
GC24	20%-30%	50%-60%	6.000
GC25	30%-40%	30%-40%	6.000
GC26	30%-40%	40%-50%	6.000
GC27	30%-40%	50%-60%	6.000
GC28	40%-50%	40%-50%	6.000
GC29	40%-50%	50%-60%	6.000
GC30	50%-60%	50%-60%	6.000

Tabela 8.4: Tempo de execução (hh:mm:ss) e consumo de memória (GB) para conjuntos de tamanho 2000, $\lambda = 1000$, usando 1, 2, 4, 8 e 16 *threads* para sequências de *GC-Content* diferentes: 20%-30%, 30%-40%, 40%-50% e 50%-60%.

Tamanho das sequências: 2.000, $\lambda = 1.000$, *threads* = 1.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:34:54	1,44	00:18:47	0,94	00:17:50	0,91	00:15:12	0,83
30–40			00:15:08	0,85	00:14:14	0,74	00:14:10	0,77
40–50					00:14:47	0,76	00:18:56	0,91
50–60							01:55:21	3,12

Tamanho das sequências: 2.000, $\lambda = 1.000$, *threads* = 2.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:17:49	1,64	00:10:51	1,06	00:10:09	1,04	00:08:49	0,93
30–40			00:10:04	0,95	00:08:37	0,84	00:09:21	0,87
40–50					00:09:04	0,86	00:10:49	1,06
50–60							00:59:01	3,52

Tamanho das sequências: 2.000, $\lambda = 1.000$, *threads* = 4.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:10:49	2,00	00:06:15	1,29	00:06:12	1,27	00:05:36	1,13
30–40			00:05:42	1,17	00:04:50	1,02	00:05:20	1,07
40–50					00:05:10	1,05	00:06:19	1,31
50–60							00:38:18	4,36

Tamanho das sequências: 2.000, $\lambda = 1.000$, *threads* = 8.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:08:52	2,76	00:05:01	1,76	00:04:37	1,75	00:04:21	1,54
30–40			00:04:15	1,58	00:03:39	1,37	00:04:05	1,46
40–50					00:03:56	1,41	00:05:09	1,81
50–60							00:33:19	6,20

Tamanho das sequências: 2.000, $\lambda = 1.000$, *threads* = 16.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:09:23	4,26	00:04:49	2,66	00:04:37	2,69	00:04:13	2,33
30–40			00:04:01	2,40	00:03:32	2,09	00:03:56	2,22
40–50					00:03:51	2,14	00:05:23	2,84
50–60							00:37:18	10,25

Tabela 8.5: Tempo de execução (hh:mm:ss) e consumo de memória (GB) para conjuntos de tamanho 3000, $\lambda = 1000$, usando 1, 2, 4, 8 e 16 *threads* para sequências de *GC-Content* diferentes: 20%-30%, 30%-40%, 40%-50% e 50%-60%.

Tamanho das sequências: 3.000, $\lambda = 1.000$, *threads* = 1.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	02:14:57	2,19	01:04:15	1,07	00:50:48	0,87	00:50:24	0,84
30–40			00:49:56	0,87	00:49:32	0,80	00:45:43	0,81
40–50					00:49:27	0,79	01:12:49	1,08
50–60							04:06:59	2,84

Tamanho das sequências: 3.000, $\lambda = 1.000$, *threads* = 2.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	01:06:54	2,49	00:33:18	1,21	00:28:33	0,98	00:26:41	0,96
30–40			00:26:05	0,98	00:27:27	0,90	00:26:38	0,92
40–50					00:26:42	0,90	00:36:42	1,26
50–60							01:54:01	3,26

Tamanho das sequências: 3.000, $\lambda = 1.000$, *threads* = 4.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:40:34	3,06	00:20:04	1,48	00:16:35	1,20	00:16:26	1,16
30–40			00:16:06	1,19	00:15:07	1,11	00:15:12	1,13
40–50					00:14:52	1,11	00:20:45	1,61
50–60							01:10:13	4,08

Tamanho das sequências: 3.000, $\lambda = 1.000$, *threads* = 8.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:34:04	4,18	00:15:43	2,03	00:12:28	1,62	00:12:57	1,57
30–40			00:12:43	1,61	00:10:25	1,52	00:11:59	1,54
40–50					00:11:58	1,52	00:17:16	2,30
50–60							01:02:57	5,82

Tamanho das sequências: 3.000, $\lambda = 1.000$, *threads* = 16.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	00:35:40	6,38	00:15:33	3,10	00:12:40	2,46	00:12:44	2,38
30–40			00:12:19	2,44	00:11:09	2,31	00:11:47	2,36
40–50					00:11:38	2,34	00:18:59	3,64
50–60							01:11:13	9,32

Tabela 8.6: Tempo de execução (hh:mm:ss) e consumo de memória (GB) para conjuntos de tamanho 6000, $\lambda = 1000$, usando 1, 2, 4, 8 e 16 *threads* para sequências de *GC-Content* diferentes: 20%-30%, 30%-40%, 40%-50% e 50%-60%.

Tamanho das sequências: 6.000, $\lambda = 1.000$, *threads* = 1.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	08:59:45	1,83	06:41:24	1,26	05:06:25	0,93	04:27:33	0,90
30–40			04:50:31	0,89	04:24:04	0,83	04:07:52	0,89
40–50					04:44:38	0,88	05:30:06	1,06
50–60							18:57:00	4,75

Tamanho das sequências: 6.000, $\lambda = 1.000$, *threads* = 2.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	04:13:40	2,04	03:11:03	1,41	02:23:53	1,05	02:17:14	1,02
30–40			02:28:04	1,00	02:16:14	0,93	02:05:39	1,02
40–50					02:24:22	1,01	02:41:44	1,24
50–60							08:41:27	5,33

Tamanho das sequências: 6.000, $\lambda = 1.000$, *threads* = 4.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	02:26:05	2,54	01:47:52	1,72	01:21:23	1,29	01:17:28	1,25
30–40			01:19:23	1,22	01:12:03	1,14	01:17:38	1,27
40–50					01:11:59	1,25	01:30:26	1,57
50–60							05:08:15	6,61

Tamanho das sequências: 6.000, $\lambda = 1.000$, *threads* = 8.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	01:56:59	3,51	01:23:50	2,32	01:01:37	1,76	01:00:25	1,70
30–40			00:58:46	1,66	00:54:03	1,55	00:57:34	1,76
40–50					00:57:07	1,75	01:12:19	2,23
50–60							04:36:29	9,41

Tamanho das sequências: 6.000, $\lambda = 1.000$, *threads* = 16.

GC \ GC	20–30		30–40		40–50		50–60	
	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.	Tempo	Mem.
20–30	02:06:57	5,45	01:26:20	3,51	01:02:04	2,68	01:00:34	2,60
30–40			00:58:25	2,56	00:53:02	2,36	01:00:38	2,68
40–50					00:59:23	2,73	01:20:00	3,49
50–60							05:13:43	15,00

As tabelas também mostram uma grande diferença no tempo total de execução para sequências de mesmo tamanho. Considerando a execução com 1 *thread* e $\lambda = 1.000$, a comparação das sequências de tamanho 2.000 (Tabela 8.4) demorou de 14 minutos a 1 hora e 55 minutos. Os tempos de execução para a comparação das sequências de 3.000 nucleotídeos (Tabela 8.5) variaram de 45 minutos a mais de 4 horas. Finalmente, entre as sequências de 6.000 nucleotídeos (Tabela 8.6) foi obtido o tempo compreendido entre 4 horas e 24 minutos e 18 horas e 57 minutos.

Ao analisar a redução no tempo de execução, as Tabelas 8.4 a 8.6 mostram que sequências longas possuem um melhor *speedup* do que as sequências menores. As sequências com 6.000 nucleotídeos apresentaram um speedup de $3,44\times$ para 4 *threads* e $4,42\times$ para 8 *threads*, enquanto as sequências de 2.000 nucleotídeos possuem um *speedup* de $2,90\times$ e $3,69\times$, respectivamente. Em todos os testes, os resultados mostraram que o uso de 16 *threads* não tem um grande impacto na redução do tempo total, mas aumenta o consumo de memória.

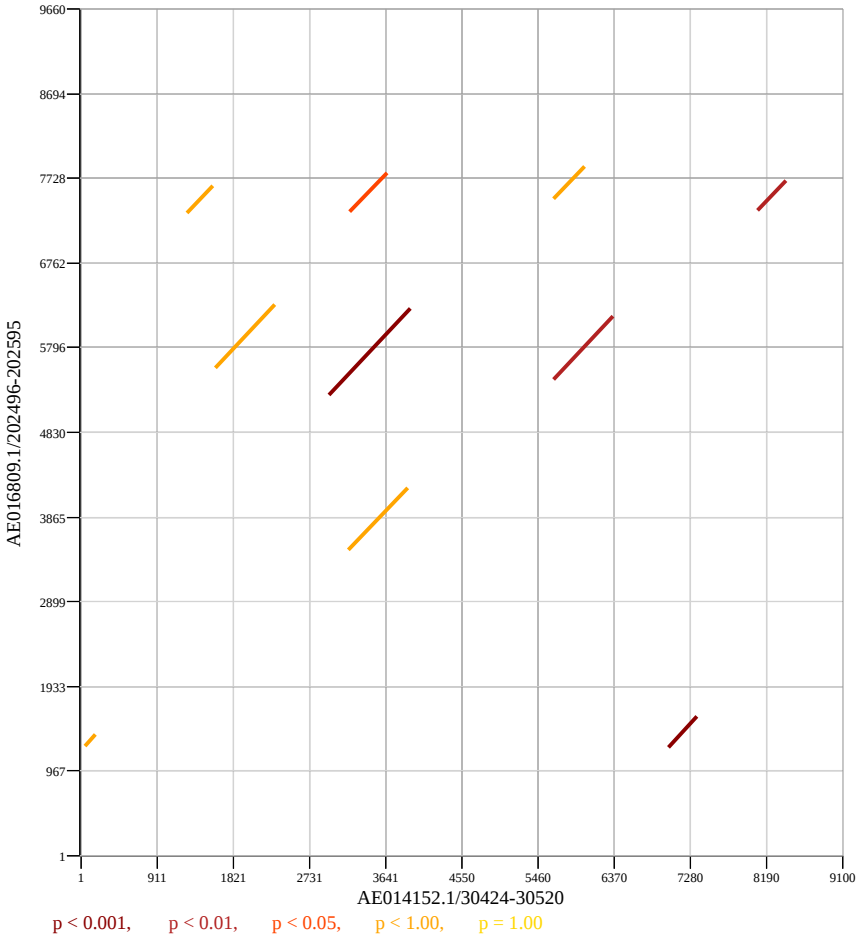
Nos casos mais demorados, a redução no tempo total de execução foi de 18 horas e 57 minutos para 4 horas e 36 minutos, com um aumento de memória de 4,75 GB para 9,41 GB. Essa redução de tempo é considerável e reforça a importância da versão *multithreaded* do Foldalign.

8.2.6 Atualização do *Web Server*

O Foldalign também possuia um *web server* disponível (Seção 3.3.5.1). A nova versão do Foldalign 2.5 foi utilizada para criar uma nova versão do *web server* disponível no endereço: `http://rth.dk/resources/foldalign/server/`. Com a atualização dos servidores para estações com vários núcleos, a versão *multithreaded* do Foldalign é importante para explorar melhor os recursos computacionais disponíveis na estação.

Originalmente, o *web server* permitia o alinhamento estrutural para sequências de 200 nucleotídeos [40]. A nova versão do servidor que executa o Foldalign 2.5 permite até 2.000 nucleotídeos. Porém, usuários que estejam autenticados dentro da rede do laboratório RTH da University of Copenhagen podem submeter sequências de tamanho até 10.000.

A Figura 8.5 ilustra o resultado de execução do *web server* do Foldalign 2.5 para um par de sequências de 9.100 a 9.660 nucleotídeos (*Synth*4). Nela, as regiões em vermelho mais intenso obtém alinhamentos com um p-value baixo, indicando menor probabilidade de serem aleatórios e maior confiabilidade de resultados. Para esse exemplo, o resultado completo está disponível no endereço: `http://rth.dk/resources/foldalign/doc/longscanexample.html`



Download pdf file of the p-value plot: foldalign.11111516151795311546 8.pdf
Download the main FOLDALIGN output file: foldalign.1111151615179531154 68.col.gz
Download the list of foldalignments: foldalign.1111151615179531154 68.hitlist

Name	Start	End	Name	Start	End	Score	P-Score	Rank
AE014152.1/30424-30520	2978	3940	AE016809.1/202496-202595	5255	6241	5394	0.000	1
AE014152.1/30424-30520	7020	7367	AE016809.1/202496-202595	1237	1587	913	0.000	2
AE014152.1/30424-30520	5647	6367	AE016809.1/202496-202595	5426	6141	829	0.001	3
AE014152.1/30424-30520	8088	8422	AE016809.1/202496-202595	7354	7690	716	0.008	4
AE014152.1/30424-30520	3220	3660	AE016809.1/202496-202595	7341	7778	682	0.018	5
AE014152.1/30424-30520	61	184	AE016809.1/202496-202595	1248	1371	634	0.055	6
AE014152.1/30424-30520	1268	1589	AE016809.1/202496-202595	7319	7640	630	0.060	7
AE014152.1/30424-30520	5654	6031	AE016809.1/202496-202595	7485	7863	622	0.071	8
AE014152.1/30424-30520	1608	2322	AE016809.1/202496-202595	5562	6278	613	0.087	9
AE014152.1/30424-30520	3206	3916	AE016809.1/202496-202595	3482	4192	591	0.141	10
AE014152.1/30424-30520	2776	2890	AE016809.1/202496-202595	9203	9317	555	0.296	11
AE014152.1/30424-30520	2804	3283	AE016809.1/202496-202595	1848	2334	544	0.364	12
AE014152.1/30424-30520	1173	1814	AE016809.1/202496-202595	8703	9343	524	0.513	13
AE014152.1/30424-30520	5337	5723	AE016809.1/202496-202595	411	789	520	0.546	14
AE014152.1/30424-30520	8122	8571	AE016809.1/202496-202595	1175	1618	494	0.764	15

Figura 8.5: Exemplo de uso do *web server* do Foldalign 2.5.

8.3 Conclusão do Capítulo

Neste capítulo, foi proposto e avaliado o Foldalign 2.5, uma nova versão *multithreaded* do algoritmo Foldalign para a obtenção do alinhamento estrutural de sequências em múltiplos *cores*. Essa nova versão foi cuidadosamente planejada para manter todas as funcionalidades existentes em versões anteriores.

Os resultados foram obtidos com sequências sintéticas, reais e aleatórias. Eles mostram que o Foldalign produz resultados com a mesma acurácia das versões anteriores, mas utilizando apenas uma fração de tempo, sendo capaz de reduzir o tempo total de execução de 8 horas e 57 minutos para 4 horas e 36 minutos. Foi também lançada uma nova versão de *web server* disponível na Internet para utilizar o Foldalign sem a necessidade de baixar e compilar o código-fonte manualmente.

Capítulo 9

CUDA-Sankoff: Estratégia em GPU para Alinhamento Estrutural Ótimo em Pares de Sequências de RNA

O CUDA-Sankoff [122] é a terceira contribuição desta tese. A principal ideia da estratégia é explorar em dois níveis o padrão de *wavefront* da matriz de programação dinâmica (DP) 4D para processar paralelamente diversas células. O algoritmo de Sankoff (Seção 3.3.2) possui uma alta complexidade computacional e normalmente é utilizado com heurísticas que não garantem o resultado ótimo. Nesta Tese, pretendemos calcular o escore do alinhamento estrutural ótimo em pares de sequências de RNA. Para poder obter resultados em tempo hábil, propomos o uso de GPU para calcular em paralelo as células da matriz de programação dinâmica 4D. A nosso conhecimento, não existe na literatura estratégia para execução do algoritmo de Sankoff em GPU.

O objetivo deste capítulo é, portanto, detalhar o projeto do CUDA-Sankoff. O projeto da estratégia e o algoritmo em GPU são detalhados na Seção 9.1. Os ambiente de testes e os resultados experimentais são descritos na Seção 9.2. Ao final, é apresentada uma discussão sobre o CUDA-Sankoff na Seção 9.3.

9.1 Projeto do CUDA-Sankoff

Existem dois grandes desafios enfrentados ao implementar o algoritmo de Sankoff em GPUs. Primeiro, a complexidade de memória é $O(L^4)$ onde L é o tamanho da maior sequência. A equação de recorrência de Sankoff possui um padrão de dependência de dados que envolve diversas células já calculadas. Além de ser um padrão complexo, o número de células a serem consultadas cresce à medida que a matriz é calculada. Neste cenário, os acessos a memória devem ser projetados com cuidado para favorecerem a localidade

enquanto são mantidas as dependências de dados. Em segundo lugar, a complexidade de tempo é $O(L^6)$ e, devido ao complexo padrão de dependência de dados, não existe um paralelismo uniforme. O paralelismo começa pequeno e atinge o seu máximo aproximadamente em 3/4 da computação da matriz 4D e então diminui novamente. Explorar esse padrão em GPU pode ser uma tarefa árdua, afinal um grande número de *threads* da GPU devem estar ativas para um melhor desempenho. Conforme explicado no Capítulo 3, as diferentes implementações baseadas em Sankoff utilizam variantes da mesma equação de recorrência. Neste trabalho, utilizamos a equação de recorrência utilizada pelo Foldalign (Seção 3.3.5.1) e FoldalginM (Seção 3.3.5.4) (Equação 3.8) e o objetivo é obter o escore ótimo com o algoritmo de Sankoff.

9.1.1 Linearização da Matriz de Programação Dinâmica do Sankoff

Para a solução do primeiro desafio, decidimos linearizar a matriz de programação dinâmica 4D. Para isso, precisamos resolver dois problemas: (a) determinar o tamanho total da matriz e alocar o seu espaço; e (b) determinar uma função de mapeamento, que recebe quatro coordenadas (i, j, k, l) e retorna uma posição única na matriz linearizada.

9.1.1.1 Cálculo do Tamanho Total e Alocação da Matriz

A Figura 9.1 mostra a matriz de programação dinâmica do Sankoff. A matriz à esquerda foi chamada de matriz externa, com índices i e k. Cada célula dessa matriz é uma outra matriz bidimensional com índices j e l, chamada de matriz interna IM. A matriz interna $IM_{4,3}$ está ilustrada à direita da Figura 9.1 e possui 6 células. Para calcular

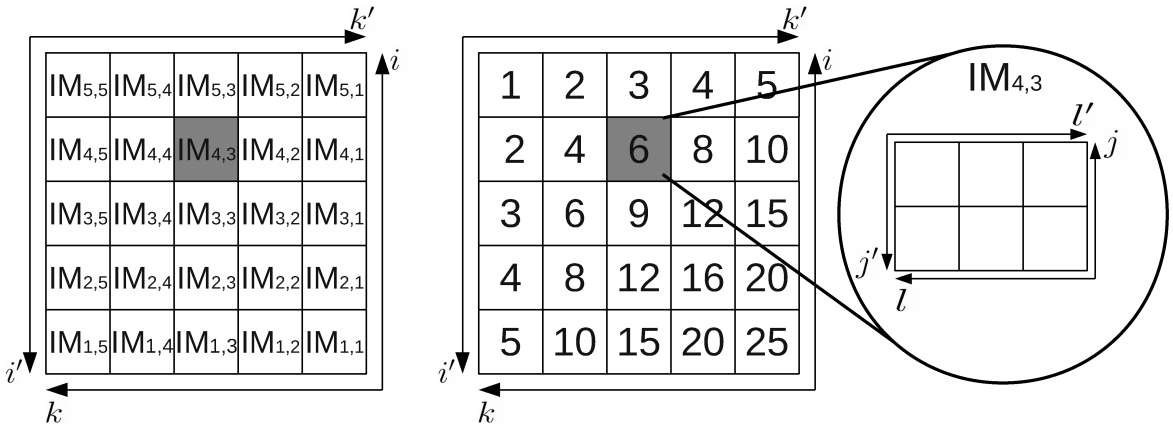


Figura 9.1: Representação gráfica da matriz de programação dinâmica 4D do algoritmo de Sankoff com os eixos i, j, k, l e i', j', k', l'.

o número total de células da matriz 4D DP, utilizar os valores de i, j, k, l não é apropriado. A figura mostra que o tamanho das matrizes internas é inversamente proporcional aos valores de i e k. No CUDA-Sankoff, definimos índices auxiliares i', j', k', e l' conforme a Equação 9.1.

$$\begin{aligned} i' &= L_A - i + 1 \\ j' &= L_A - j + 1 \\ k' &= L_B - k + 1 \\ l' &= L_B - l + 1 \end{aligned} \tag{9.1}$$

Desta forma, cada célula da matriz de programação dinâmica possui uma coordenada positiva (i', j', k', l') tal que $1 \leq i' \leq L_A, 1 \leq j' \leq i', 1 \leq k' \leq L_B, 1 \leq l' \leq k'$. Essas funções auxiliares são úteis para calcular o tamanho total da matriz. Como pode ser visto na Figura 9.1, o tamanho das matrizes internas é diretamente proporcional aos valores de i' e k'. Afinal, cada matriz interna possui i' linhas e k' colunas, com tamanho total $i' \cdot j'$, como mostrado na Equação 9.2.

$$|IM_{i',k'}| = i' \cdot k' \tag{9.2}$$

É importante notar que o número de elementos de uma matriz interna na mesma coluna é um múltiplo do número de elementos da matriz interna que está na primeira linha, definido de acordo com a Equação 9.3.

$$\begin{aligned} |IM_{i',k'}| &= i' \cdot k' \\ &= i' \cdot (1 \cdot k') \\ &= i' \cdot |IM_{1,k'}| \end{aligned} \tag{9.3}$$

Considere agora que $|R_n|$ é o tamanho da n-ésima linha da matriz S de programação dinâmica 4D. A matriz S possui L_A linhas e L_B colunas, onde L_A e L_B são o tamanho da primeira e segunda sequência, respectivamente. Portanto, a primeira linha R_1 possui

L_B colunas e o seu tamanho $|R_1|$ é dado pela Equação 9.4.

$$\begin{aligned} |R_1| &= |IM_{1,1}| + |IM_{1,2}| + |IM_{1,3}| + \cdots + |IM_{1,L_B}| \\ &= 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 3 + \cdots + 1 \cdot L_B \\ &= \sum_{x=1}^{L_B} x \\ &= \frac{(1 + L_B) \cdot L_B}{2} \end{aligned} \tag{9.4}$$

Usando a equação 9.3 é possível ver que o tamanho da n-ésima coluna é um múltiplo do tamanho da primeira coluna, conforme mostra a Equação 9.5.

$$\begin{aligned} |R_n| &= |IM_{n,1}| + |IM_{n,2}| + \cdots + |IM_{n,L_B}| \\ &= n \cdot |IM_{1,1}| + n \cdot |IM_{1,2}| + \cdots + n \cdot |IM_{1,L_B}| \\ &= n \cdot (|IM_{1,1}| + |IM_{1,2}| + \cdots + |IM_{1,L_B}) \\ &= n \cdot |R_1| \end{aligned} \tag{9.5}$$

Finalmente, é possível calcular o tamanho total da matriz S ($|S|$) como um somatório do tamanho das suas L_A linhas (Equação 9.6):

$$\begin{aligned} |S| &= |R_1| + |R_2| + |R_3| + \cdots + |R_{L_A}| \\ &= 1 \cdot |R_1| + 2 \cdot |R_1| + 3 \cdot |R_1| + \cdots + L_A \cdot |R_1| \\ &= |R_1| \cdot (1 + 2 + 3 + \cdots + L_A) \\ &= \Big(\sum_{x=1}^{L_B} x\Big) \cdot \Big(\sum_{y=1}^{L_A} y\Big) \\ &= \Big(\frac{(1 + L_B) \cdot L_B}{2}\Big) \cdot \Big(\frac{(1 + L_A) \cdot L_A}{2}\Big) \end{aligned} \tag{9.6}$$

Usando a Equação 9.6, temos que a primeira célula da matriz $IM_{4,4}$ corresponde à célula 130 na matriz linearizada.

9.1.1.2 Função de Mapeamento

Com o espaço total da matriz alocado em memória, definimos uma função de mapeamento que recebe como argumento as coordenadas i, j, k, l da matriz S e retorna uma posição única. Para isso, dividimos a função de mapeamento em três variáveis $\Delta_{i'}$, $\Delta_{k'}$ e Δ_{IM}, que dependem da linha, coluna e tamanho da matriz interna, respectivamente.

A Figura 9.2 ilustra um exemplo de mapeamento para as células que estejam dentro da matriz interna $IM_{4,4}$. A primeira parte é $\Delta_{i'}$, em cinza claro, e representa o somatório

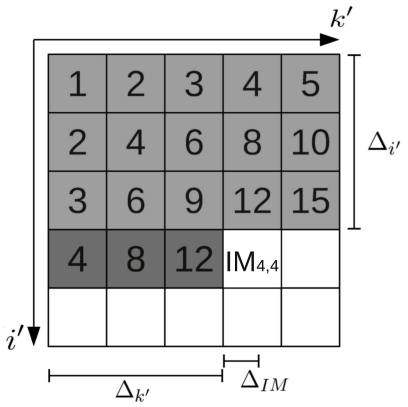


Figura 9.2: Representação gráfica ilustrando o $\Delta_{i'}$, $\Delta_{k'}$ e Δ_{IM} para as células da matriz interna $IM_{i',k'} = IM_{4,4}$.

do tamanho de todas as linhas anteriores. Para isso, devemos efetuar o somatório do tamanho de todas as linhas x tal que $1 \leq x \leq i' - 1$, conforme a Equação 9.7.

$$\begin{aligned}
\Delta_{i'} &= \sum_{x=1}^{i'-1} R_x \\
&= |R_1| + |R_2| + |R_3| + \cdots + |R_{i'-1}| \\
&= |R_1| + 2 \cdot |R_1| + 3 \cdot |R_1| + \cdots + (i' - 1) \cdot |R_1| \\
&= |R_1| \cdot (1 + 2 + \cdots + (i' - 1)) \\
&= \left(\frac{(1 + L_B) \cdot L_B}{2}\right) \cdot \left(\frac{(1 + (i' - 1)) \cdot (i' - 1)}{2}\right) \\
&= \left(\frac{(1 + L_B) \cdot L_B}{2}\right) \cdot \left(\frac{i' \cdot (i' - 1)}{2}\right)
\end{aligned} \tag{9.7}$$

A segunda parte é $\Delta_{k'}$, representada em cinza escuro na Figura 9.2. Ela é baseada no valor k' e representa a soma do tamanho de todas as matrizes internas da linha i' e

colunas de 1 a $k' - 1$, ilustrada na Equação 9.8.

$$\begin{aligned}
\Delta_{k'} &= \sum_{x=1}^{k'-1} IM_{i',x} \\
&= |IM_{i',1}| + |IM_{i',2}| + |IM_{i',3}| + \cdots + |IM_{i',k'-1}| \\
&= i' \cdot 1 + i' \cdot 2 + i' \cdot \cdot 3 + \cdots + i' \cdot (k' - 1) \\
&= i' \cdot (1 + 2 + 3 + \cdots + (k' - 1)) \\
&= i' \cdot \left(\frac{(1 + (k' - 1)) \cdot (k' - 1)}{2} \right) \\
&= i' \cdot \left(\frac{k' \cdot (k' - 1)}{2} \right)
\end{aligned} \tag{9.8}$$

A última parte, Δ_{IM} é utilizada para distinguir células que pertencem a mesma matriz interna, ou seja, é uma posição única na matriz interna $IM_{i',k'}$. Toda matriz interna $IM_{i',k'}$ é uma matriz bidimensional com k' colunas e i' linhas, com valores distintos de j' e l', tal que $0 < j' < i'$ e $0 < l' < k'$. Portanto, podemos utilizar a fórmula padrão de linearização de matrizes bidimensionais, conforme ilustrado na Equação 9.9.

$$\Delta_{IM} = (j' - 1) \cdot k' + (l' - 1) \tag{9.9}$$

Portanto, definimos a equação de mapeamento, como o somatório das Equações 9.7, 9.8 e 9.9, conforme a Equação 9.10.

$$\begin{aligned}
&f(i', j', k', l') = \Delta_{i'} + \Delta_{k'} + \Delta_{IM} \\
&= \left(\frac{(1 + L_B) \cdot L_B}{2} \right) \cdot \left(\frac{i' \cdot (i' - 1)}{2} \right) + i' \cdot \left(\frac{k' \cdot (k' - 1)}{2} \right) + (j' - 1) \cdot k' + (l' - 1)
\end{aligned} \tag{9.10}$$

9.1.2 Algoritmo Paralelo Proposto

Para executar o algoritmo de Sankoff, optamos pelas GPUs devido ao seu paralelismo massivo. O algoritmo paralelo é baseado no fato de que matrizes internas que pertencem à mesma diagonal podem ser processadas em paralelo (padrão *wavefront*). Dada uma matriz interna IM, células que pertencem à mesma diagonal dessa matriz interna também podem ser processadas em paralelo, seguindo também o padrão *wavefront*. Para explorar o paralelismo, é proposto um algoritmo paralelo baseado em Sankoff (Seção 3.3.2) que processa tanto as diagonais da matriz externa com da matriz interna segundo o padrão de *wavefront* de exploração de paralelismo, conforme ilustrado nos Algoritmos 11 e 12.

Algoritmo 11 Diagonal_Sankoff

```
1: for external_diag in S do
2:    for IM in external_diag do
3:       EXPAND_IM(A, B, IM)
4:    end for
5: end for
```

Algoritmo 12 EXPAND_IM(A, B, IM)

```
1: i' ←get_i(IM)
2: k' ←get_k(IM)
3: for internal_diag in IM do
4:    for cell in internal_diag do
5:       j' ←get_j(cell)
6:       l' ←get_l(cell)
7:       EXPAND_POSITION(A, B, i', j', k', l')
8:    end for
9: end for
```

No Algoritmo 11, as diagonais da matriz externa S são processadas uma a uma (linha 1). Cada matriz interna que pertence à diagonal externa que está sendo calculada é expandida em paralelo (linhas 2 e 3).

O procedimento EXPAND_IM (Algoritmo 12) calcula todas as células de uma matriz interna IM. Inicialmente, os índices i' e k' da matriz IM são obtidos (linhas 1 e 2). As funções get_i(IM) e get_k(IM) retornam os índices i' e k' da matriz interna $IM_{i',k'}$. Após isso, as diagonais da matriz interna IM são calculadas diagonal a diagonal (linha 3). Para cada célula da diagonal interna que está sendo calculada, as funções auxiliares get_j($cell$) e get_l($cell$) retornam os índices j' e l' da célula que será processada. Depois disso, o procedimento EXPAND_POSITION (linha 7) computa a Equação 3.11 para uma célula de coordenada (i', j', k', l'), calculando os escores baseados em *matches*, *mismatches*, pares de bases, substituições compensatórias e também aplicando a regra de *multibranch*, quando duas subestruturas são agrupadas para criar uma maior.

A Equação de recorrência do algoritmo de Sankoff (Equação 3.8) mostra que ao processar uma matriz interna IM com coordenadas (i', k'), os valores das matriz $IM_{i-1,k}$, $IM_{i,k-1}$ e $IM_{i-1,k-1}$ precisam ser lidos. Então, células que estão na mesma diagonal podem ser processadas em *waverfront*. A Figura 9.3 ilustra o padrão *wavefront* usado na estratégia proposta, onde células com a mesma cor podem ser processadas em paralelo. No detalhe à direita são mostradas as células de uma matriz interna que podem ser calculadas em paralelo.

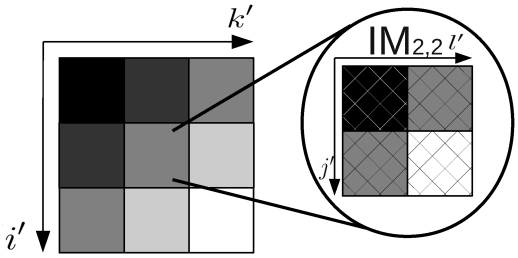


Figura 9.3: Processamento de *waverfront* externo (esquerda) e interno (direita) do algoritmo paralelo.

9.1.3 Algoritmo em GPU

Algoritmo 13 CUDA_Diagonal_Sankoff(A, B)

```
1: for external_diag in S do
2:     th ←get_th(external_diag)
3:     im ←count_im(external_diag)
4:     GPU_EXPAND_IM <<< im, th >>> (A, B, S, external_diag)
5: end for
```

Algoritmo 14 GPU_EXPAND_IM($A, B, S, external\_diag$)

```
1: i' ←get_i(blockIdx.x, external_diag)
2: k' ←get_k(blockIdx.x, external_diag)
3: IM ←get_im(i', k')
4: for id in IM do
5:     j' ←get_j(threadIdx.x, id)
6:     l' ←get_l(threadIdx.x, id)
7:     GPU_EXPAND_POSITION(A, B, S, i', j', k', l')
8:     __syncthreads()
9: end for
```

Os Algoritmos 13 e 14 ilustram a solução em GPU baseada no algoritmo descrito na Seção 9.1.2. Esses algoritmos assumem que a matriz de programação dinâmica S está previamente alocada em memória global. Todas as diagonais da matriz externa são processadas uma a uma. O Algoritmo 13 ilustra o CUDA-Sankoff. Na CPU, para cada diagonal da matriz externa (linha 1), funções auxiliares são utilizadas para se obter o número de matrizes internas que serão calculadas e de *threads* que serão criadas para calcular cada diagonal da matriz externa *external_diag* (linhas 2 e 3). Desta forma, são criados *im* blocos com *th threads*, onde *im* é o número de matrizes internas que a diagonal possui, e *th* é o maior número de *threads* necessário para processar as matrizes internas. Na linha 4, o *kernel* (Seção 4.4.2) da GPU é invocado.

Na função de *kernel* GPU_EXPAND_IM (Algoritmo 14), cada bloco é responsável por calcular uma matriz interna. Funções auxiliares são usadas para obter os índices i' e k' (linhas 1 e 2), baseada no blockId e a contagem de diagonal já processadas. Para cada diagonal da matriz interna (linhas 4), as posições j' e l' são calculadas baseadas no *threadId* e no contador do número de diagonais da matriz interna já processados (linhas 5 e 6), então a célula de índice i', j', k', l' é processada (linha 7). A função GPU_EXPAND_POSITION calcula todos os membros da equação de recorrência para uma célula. Devido à dependência de dados, todas as *threads* devem ser sincronizadas antes de calcular a próxima diagonal (linha 8).

9.2 Resultados Experimentais

Nesta Tese, foram implementadas duas versões do CUDA-Sankoff: uma para CPU (Algoritmo 11) e uma para GPU (Algoritmo 13). A solução em CPU foi implementada em C++ e OpenMP. Nesta solução um *pragma* foi adicionado para paralelizar os *loops* do Algoritmo 11 (linha 2) e do Algoritmo 12 (linha 4). Neste versão, foi utilizada a opção OpenMP de escalonamento dinâmico pois, através de testes, foi determinado empiricamente que este tipo de escalonamento possui o melhor desempenho. Isso pode ser explicado devido ao fato das matrizes internas que estão na mesma diagonal possuírem tamanhos e tempo de processamento diferentes.

Alguns métodos baseados no algoritmo de Sankoff, como Foldalign (Seção 3.3.5.1), utilizam modelo de energia para calcular as interações entre os pares de bases, levando a um sistemas de escore diferente da equação de recorrência implementada no CUDA-Sankoff. Além disso, essas implementações utilizam heurísticas para reduzir a complexidade de tempo e memória, porém o resultado ótimo não é mais garantido. Desta forma, não é possível comparar o tempo de execução entre o CUDA-Sankoff e os métodos encontrados no estado da arte. Por isso, optou-se por comparar o tempo de execução da CPU e GPU do mesmo algoritmo. A solução em GPU implementa os Algoritmos 13 e 14 em CUDA C, utilizando o CUDA toolkit 7.5.18.

9.2.1 Ambiente de testes e Sequências Utilizadas

Os experimentos foram executados em uma máquina com 2 processadores Intel Xeon E5-2650, com 2,00 GHz (8 *cores*) e 64 GB de RAM, para a versão em CPU. A versão em GPU foi testada em duas GPUs diferentes: NVidia GTX 780, com 2.304 CUDA *cores*, 900 MHz e 2 GB de memória, da arquitetura Kepler (Seção 4.4.3) e NVidia 980 Ti, com 2.816 CUDA *cores*, 1.075 MHz e 6 GB de memória RAM, da arquitetura Maxwell (Seção 4.4.4).

Tabela 9.1: Famílias de sequências utilizadas nos testes e seu tamanho médio.

Família RFAM	Descrição	# Seq	Tamanho Médio
RF01414	Class I non-coding RNA	2	46
RF00981	mir-939 MicroRNA	2	81
RF01735	SAM-I-IV variant riboswitch	2	100
RF01999	group II D1D4-2 intron	2	120
RF00290	BaMV cis-regularoty-element	2	140
RF02053	STnc430 Bacterial small RNA	2	160
RF01543	LSU_rRNA_eukarya	2	180
RF01788	drz-agan-2-2-riboswitch	2	201
RF02079	STnc180 Bacterial small RNA	2	214
RF01845	enod 40 non-coding RNA	2	239
RF02365	Cyanobacterial functional RNA 17	2	261
RF02514	5' ureB small RNA	2	281

A Tabela 9.1 apresenta as famílias de sequência utilizadas nos testes, informando seu identificador RFAM, seu nome e tamanho médio das sequências que as compõem. Para cada família, foram escolhidas de maneira aleatória duas sequências de tamanho similar. Como pode ser visto, o tamanho médio das sequências varia de 46 a 281 nucleotídeos.

9.2.2 Tempo de Execução e *speedup*

A Tabela 9.2 mostra o tempo de execução da soluções em CPU usando 1 e 16 *threads*, o tempo de execução em duas GPUs, *speedups* obtidos das GPUs em relação às CPUs e a Figura 9.4 mostra os speedups obtidos no experimento comparando as duas GPUs e a versão em CPU executando em 16 *cores*. Nos nossos testes, a solução da CPU serial, da CPU com 16 *threads* e da GPU obtiveram o mesmo resultado para os mesmos conjuntos de sequências. Isso é o esperado pois elas implementam a mesma equação de recorrência.

Pode ser visto na Tabela 9.2 que a solução em GPU não obtém um bom tempo de execução para as sequências de tamanho médio 46 (RF01414). A execução em 16 *cores* leva menos de um segundo, enquanto a solução em GPU demora mais de dois segundos. Isto acontece principalmente pelo *overhead* de invocação do *kernel* e transferências de memória de/para a GPU.

Para as sequências de tamanho médio 81 (RF00981) a 201 (RF02543), os *speedups* da GPU em relação à CPU de 16 *cores*, cresce constantemente atingindo *speedups* de 13,24x e 21,88x para as GPUs GTX 780 e GTX 980 Ti, respectivamente. Para a GTX 980 Ti, os speedups continuam a crescer até 24,13x para sequências de tamanho médio 281 (RF02514). Observa-se um crescimento menor do *speedup* na comparação das sequências de tamanho médio 239, 261 e 281. Neste caso, acreditamos ter atingido o paralelismo máximo da GTX 980 Ti. O mesmo comportamento acontece para a GTX 780 para as

Tabela 9.2: Comparação do tempo de execução do Sankoff em CPU serial (1CPU), em CPU 16 *threads* (16CPU) e em duas GPU (GPU 780 e GPU 980 Ti).

Sequência	1CPU hh:mm:ss	16CPU hh:mm:ss	GPU 780 hh:mm:ss	Speedup 16 CPU	GPU 980 Ti hh:mm:ss	Speedup 16 CPU	Mem. (GB)
RF01414	00:00:05	00:00:01	00:00:02	0,44x	00:00:02	0,43x	<0,01
RF00981	00:02:47	00:00:22	00:00:18	1,21x	00:00:15	1,45x	0,04
RF01725	00:11:11	00:01:18	00:00:42	1,85x	00:00:35	2,23x	0,09
RF01999	00:39:43	00:05:05	00:01:28	3,46x	00:01:12	4,19x	0,19
RF00290	01:43:24	00:16:55	00:02:49	5,99x	00:02:17	7,39x	0,36
RF02053	03:53:55	00:46:05	00:05:06	9,03x	00:03:59	11,53x	0,61
RF02543	08:32:16	01:43:59	00:08:59	11,57x	00:06:34	15,82x	0,98
RF01788	16:22:27	03:39:45	00:16:35	13,24x	00:10:47	20,36x	1,53
RF02079	24:51:02	05:16:12	**00:23:52**	**13,25x**	00:14:26	21,88x	1,95
RF01845	49:42:47	10:07:09	sem mem.	—	00:27:10	22,35x	3,06
RF02365	84:50:19	18:14:16	sem mem.	—	00:46:57	23,30x	4,35
RF02514	**129:04:10**	**28:39:19**	sem mem.	—	**01:11:13**	**24,13x**	**5,84**

sequências de tamanho 201 (RF01788) e 214 (RF02079), com *speedups* de 13,24x e 13,25x, respectivamente. Nesta placa, as execuções para as sequências das famílias RF01845, RF02365 e RF02514 ficaram sem memória e por esse motivo não foram concluídas com sucesso.

Os tempos de execução na Tabela 9.2 mostram que, como esperado, o algoritmo de Sankoff requerer uma quantidade enorme de poder computacional. A execução para sequências da família RF02514 com tamanho médio 281 demorou em um core mais de 5 dias para terminar, enquanto a execução utilizando 16 *cores* demorou mais de um dia. Finalmente, a execução na GPU GTX 980 Ti demorou 1 hora e 11 minutos. Isso mostra que as GPUs são uma alternativa muito boa para executar o algoritmo de Sankoff.

9.3 Conclusão do Capítulo

Neste capítulo propusemos e avaliamos uma estratégia baseada em GPU para executar o algoritmo de Sankoff de modo a obter o escore ótimo do alinhamento estrutural de pares de sequências. Até onde sabemos, esta é a primeira implementação do algoritmo de Sankoff em unidades de processamento gráfico. Na nossa estratégia, usamos paralelismo multi-nível em *wavefront* e optamos por linearizar a matriz de programação dinâmica 4D.

Foram utilizados nos experimentos pares de sequências obtidos de famílias RFAM, com comprimento médio variando entre 46 e 281. Para cada comparação, foi obtido o tempo em CPU serial, CPU de 16 *cores*, uma GPU GTX 780 e uma GPU GTX 980 Ti. Os resultados mostram que o CUDA-Sankoff possui um tempo de execução melhor quando comparado a uma execução em 16 *cores* de CPU, sendo até 24 vezes mais rápido. Quando

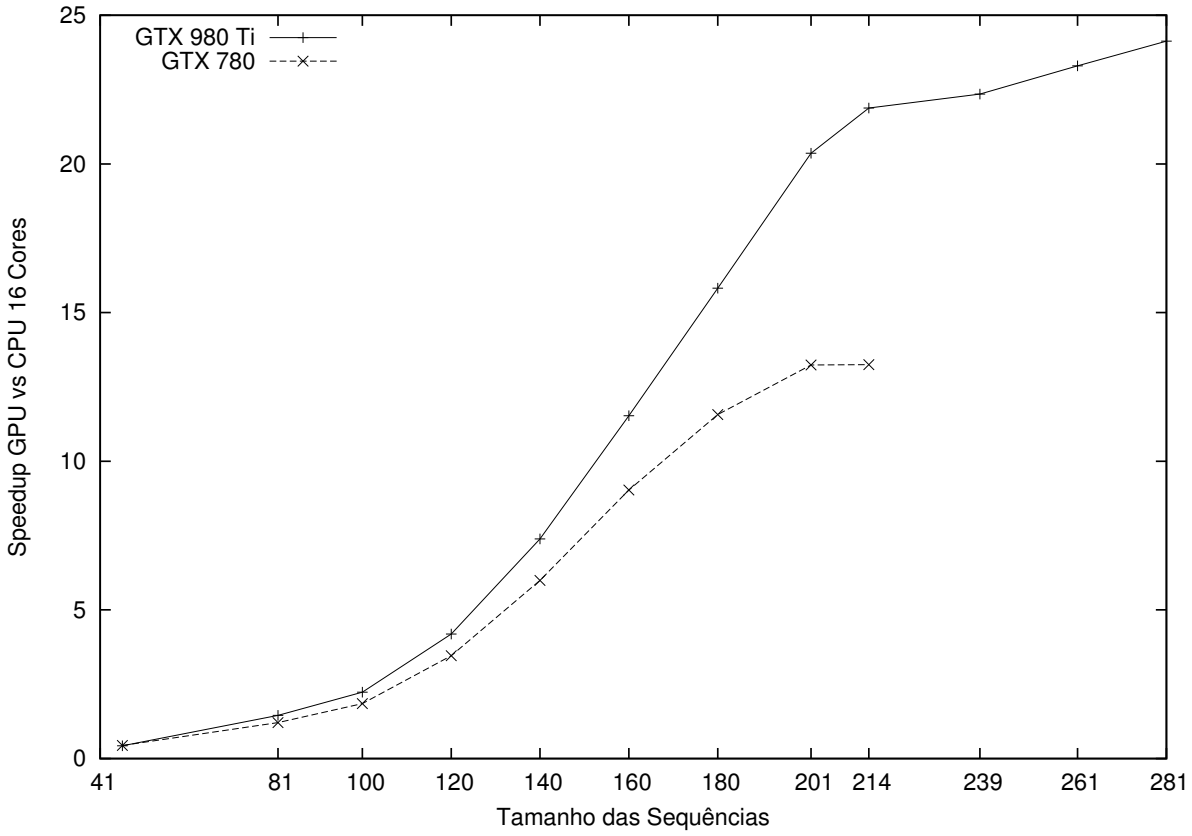


Figura 9.4: *Speedups* do CUDA-Sankoff em relação à solução CPU *multithreaded* (16 *cores*)

comparado a uma execução serial em CPU (sem instruções vetoriais), o CUDA-Sankoff é até 109 vezes mais rápido. No melhor caso, o tempo total foi reduzido de 5 dias e 9 horas para 1 hora e 11 minutos.

Parte III

Conclusão

Capítulo 10

Conclusão e Trabalhos Futuros

Nesta Tese de doutorado, foi investigado o problema do alinhamento de sequências biológicas, com focos em alinhamento primário múltiplo exato de sequências e alinhamento estrutural (secundário) heurístico e exato de sequências. Como ambos os problemas possuem alta complexidade (NP-Completo e $O(n^6)$, respectivamente), investigamos o uso de estratégias paralelas para acelerar a obtenção de resultados. Foram propostas e avaliadas três estratégias paralelas para abordar esses problemas. A primeira, chamada de PA-Star, é um algoritmo paralelo para o alinhamento primário múltiplo ótimo de sequências biológicas, reduzindo o espaço de busca com o algoritmo A-Star e foi desenvolvida para arquiteturas *multicore*. A segunda, chamada de Foldalign 2.5, é uma nova versão paralela da ferramenta Foldalign, que utiliza heurísticas para a redução da complexidade de memória e tempo e também foi desenvolvida para estações *multicore*. A terceira, chamada de CUDA-Sankoff, é uma versão paralela do algoritmo de Sankoff para o alinhamento secundário de sequências de RNA, obtendo o resultado ótimo e utilizando GPUs para reduzir o tempo de execução.

Os resultados obtidos com conjuntos de sequência do BAliBASE (referência 1) mostram que o PA-Star é capaz de reduzir o tempo de execução de $75,20$s para $7,23$s, utilizando as otimizações propostas nessa Tese: função de *hash* sensível à localidade, estrutura de dados multi-index, *templates* e abordagem *multithreaded*. O PA-Star foi capaz de comparar 79 de 82 instâncias do BAliBASE em 4 horas e 56 minutos, produzindo o alinhamento múltiplo ótimo para todos os 79 conjuntos. Foi também capaz de obter o alinhamento ótimo para um conjunto real de 22 sequências e, até onde sabemos, nenhum outro trabalho obteve um alinhamento ótimo para um número tão alto de sequências. Além disso, mostramos que o PA-Star possui um desempenho superior a outra estratégia do estado da arte, o PFA*-DDD (Seção 5.2.4), sendo capaz de executar $4,77\times$ mais rápido. Também foi mostrado que o PA-Star combinado com a estratégia de disco DAPA é capaz de obter o alinhamento múltiplo de sequências quando o PA-Star sem DAPA e o

PFA*-DDD falham por falta de memória. Finalmente, a nosso conhecimento, o PA-Star é a única solução na literatura baseada no algoritmo A-Star com, concomitantemente, alocação de trabalho a *threads* sensível à localidade e uso de memória externa (disco).

O Foldalign 2.5 proposto nesta Tese, abre a possibilidade de alinhamento estrutural heurístico de sequências muito mais longas em um tempo razoável. O Foldalign 2.5 é capaz de obter as mesmas respostas das versões anteriores, porém em tempo bastante reduzido. Comparando sequências de 6.000 nucleotídeos, o tempo foi diminuído de 8 horas e 57 minutos para 4 horas e 36 minutos. O Foldalign 2.5 é a base de uma nova versão do *web server*, com a capacidade de analisar sequências muito maiores, que está disponível na Internet.

Nesta Tese, propusemos também uma estratégia em GPU para executar o algoritmo de Sankoff e obter o alinhamento secundário ótimo em pares para as sequências de RNA. Até onde sabemos, essa é a primeira implementação do algoritmo de Sankoff para GPUs. Os resultados mostram que o CUDA-Sankoff é capaz de executar mais rapidamente que uma CPU utilizando 16 *cores*, sendo até 24× mais rápido. Quando comparado a uma CPU serial, o CUDA-Sankoff é capaz de reduzir o tempo total de 5 dias e 9 horas para apenas 1 hora e 11 minutos.

Analisando os resultados obtidos, concluímos que evoluímos o estado da arte e todos os objetivos, geral e específicos foram atingidos (Seção 1.2). As estratégias propostas nesta Tese são capazes de obter em tempo factível resultados que antes não haviam sido alcançados: obter o alinhamento primário ótimo múltiplo para 22 sequências; obter o alinhamento secundário heurístico para sequências de RNA, baseado no algoritmo de Sankoff, com quase 10.000 caracteres; e executar o algoritmo de Sankoff para pares de sequências de RNA com até 281 nucleotídeos.

10.1 Trabalhos Futuros

Nesta Tese, o PA-Star foi implementado com um ambiente *symmetric multiprocessor* (SMP) em mente. Como trabalho futuro, pretendemos estender o PA-Star para executar em arquiteturas NUMA, modificando a função de *hash* sensível à localidade para considerar a distância entre os *cores* durante a distribuição do espaço de busca.

Além disso, a função f de prioridade do algoritmo A-Star utilizada foi $h_{2,all}$, baseada no alinhamento ótimo em pares de cada uma das sequências utilizadas. Porém, são conhecidas outras funções de avaliação, combinando mais de duas sequências. Como trabalho futuros, gostaríamos de implementar essas funções mais robustas no PA-Star, utilizando GPUs quando exigirem um alto poder computacional.

Finalmente, para o PA-Star, gostaríamos de verificar o impacto de todos os parâmetros como número de *threads*, números de região, funções de *hash*, memória RAM disponível, número de sequências e similaridade das sequências como fatores de impacto no tempo total de execução.

A nova versão do Foldalign 2.5 abre possibilidades de se procura RNAs em uma escala maior, utilizando apenas uma fração do tempo. Futuramente, planejamos fazer testes precisos da relação entre as heurísticas e os escores ótimos das sequências. Adicionalmente, gostaríamos de otimizar o código da função de *multibranch*, agrupando as células em blocos próximos e realizando operações vetoriais para obtenção do escore ótimo.

Além disso, o Foldalign é a base do algoritmo heurístico múltiplo FoldalignM (Seção 3.3.5.4). Gostaríamos de atualizar o FoldalignM para utilizar o Foldalign 2.5, explorando o paralelismo intra-sequências proposto nessa Tese. Além disso, o FoldalignM requer o alinhamento em par-a-par de todas as sequências envolvidas. Para isso, gostaríamos de criar regras de balanceamento de carga para utilizar múltiplas estações em *cluster* de computadores para explorar o paralelismo inter-sequências dessa etapa.

A primeira versão do CUDA-Sankoff obtém apenas a similaridade das sequências. Em uma versão futura, gostaríamos de obter não apenas o escore, mas também o alinhamento das sequências, adicionando a fase de *backtrace* como última etapa de execução. Adicionalmente, gostaríamos de utilizar uma função de similaridade mais relevante biologicamente, adicionando o *base-pair probabilities* do algoritmo de McCaskill (Seção 3.2.2) como função de escore de pareamento de nucleotídeos e o RIBOSUM85-60 (Seção 3.4.3) como escore para nucleotídeos não pareados e de substituição. Finalmente, foi mostrado nessa Tese que o alinhamento secundário ótimo, mesmo utilizando arquiteturas de alto desempenho, é limitado a pequenas sequências, de até 200 nucleotídeos, e utilizando apenas duas sequências. Para o alinhamento primário, ilustramos no Capítulo 2 que não é necessário percorrer todo o espaço de busca para garantir o resultado ótimo, porém não conhecemos nenhum método que garanta isso para o alinhamento secundário. Sendo assim, pretendemos como trabalho futuro investigar um algoritmo que reduza o espaço de busca do alinhamento secundário ótimo, permitindo obter o alinhamento para sequências mais longas e também para três ou mais sequências.

Referências

[1] S. F. Altschul, R. J. Carroll, and D. J. Lipman. Weights for data related by a tree. *Journal of Molecular Biology*, 207(4):647–653, 1989. 19

[2] S. F. Altschul and D. J. Lipman. Trees, stars, and multiple biological sequence alignment. *SIAM Journal on Applied Mathematics*, 49(1):197–209, 1989. 79

[3] B. Barker. Message passing interface (MPI). In *Workshop: High Performance Computing on Stampede*, volume 262, 2015. 58

[4] J. Blazewicz, W. Frohmberg, M. Kierzynka, and P. Wojciechowski. G-MSA - A GPU-based, fast and accurate algorithm for multiple sequence alignment. *Journal of Parallel and Distributed Computing*, 73(1):32–41, 2013. 78, 85

[5] U. Bondhugula, A. Acharya, and A. Cohen. The pluto+ algorithm: a practical approach for parallelization and locality optimization of affine loop nests. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(3):12, 2016. 98

[6] E. Burns, S. Lemons, W. Ruml, and R. Zhou. Best-first heuristic search for multicore machines. *J. Artif. Intell. Res. (JAIR)*, 39:689–743, 2010. 87

[7] J. J. Cannone, S. Subramanian, M. N. Schnare, J. R. Collett, L. M. D'Souza, Y. Du, B. Feng, N. Lin, L. V. Madabusi, K. M. Müller, et al. The comparative RNA web (CRW) site: an online database of comparative sequence and structure information for ribosomal, intron, and other RNAs. *BMC Bioinformatics*, 3(1):2, 2002. 128

[8] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Mathematics*, 48:1073–1082, 1988. xiv, 1, 3, 9, 16

[9] S. Cawley, S. Bekiranov, H. H. Ng, P. Kapranov, E. A. Sekinger, D. Kampa, A. Piccolboni, V. Sementchenko, J. Cheng, A. J. Williams, et al. Unbiased mapping of transcription factor binding sites along human chromosomes 21 and 22 points to widespread regulation of noncoding rnas. *Cell*, 116(4):499–509, 2004. 2

[10] K. Chaichoompu, S. Kittitornkun, and S. Tongsima. MT-ClustalW: multithreading multiple sequence alignment. In *20th International Conference on Parallel and Distributed Processing*, pages 254–254. IEEE Computer Society, 2006. xv, 72, 73, 85

[11] X. Chen, C. Wang, S. Tang, C. Yu, and Q. Zou. CMSA: a heterogeneous CPU/GPU computing system for multiple similar RNA/DNA sequence alignment. *BMC Bioinformatics*, 18(1):315, 2017. 79, 85

[12] E. P. Consortium et al. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57, 2012. 2

[13] F. Crick. The origin of the genetic code. *Journal Molecular Biology*, (38):367–379, 1968. 2, 28

[14] X. Cui and A. Palazzo. Localization of mRNAs to the endoplasmic reticulum. *Wiley Interdisciplinary Reviews: RNA*, 5(4):481–492, 2014. 2

[15] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5(suppl 3):345–351, 1978. xvii, 10

[16] E. V. Denardo. *Dynamic programming: models and applications*. Courier Corporation, 2012. 11, 31

[17] J. Dongarra. Trends in High-Performance Computing. *IEEE Circuits & Devices Magazine*, pages 22–27, 2006. 52

[18] R. Dowell and S. Eddy. Efficient pairwise RNA structure prediction and alignment using sequence alignment constraints. *BMC Bioinformatics*, 7:400, 2006. 45

[19] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999. 8, 9, 28

[20] O. Duzlevski. SMP version of ClustalW 1.82. 2002. 72

[21] R. C. Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32:1792–1797, 2004. 76

[22] M. Fabian, N. Sonenberg, and W. Filipowicz. *Annual review of biochemistry*, 79(1), 2010. 2, 28

[23] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972. 52, 60

[24] P. Gachet, B. Joinnault, and P. Quinton. Synthesizing systolic arrays using DIASTOL. In *International workshop on systolic arrays*, pages 25–36, 1986. 92

[25] P. P. Gardner and R. Giegerich. A comprehensive comparison of comparative RNA structure prediction approaches. *BMC Bioinformatics*, 5:140, 2004. xiv, 36, 37

[26] J. Gorodkin, L. Heyer, and G. Stormo. Finding the most significant common sequence and structure motifs in a set of RNA sequences. *Nucleic Acids Research*, 25(18):3724–3732, 1997. 43

[27] J. Gorodkin and W. L. Ruzzo. *RNA Sequence, Structure and Function: Computational and Bioinformatic Methods*. Humana Press, 2014. xiv, xv, 2, 28, 29, 32, 38, 41, 42

[28] A. Gregory and Z. Wang. A day in the life of the spliceosome. *Nature Reviews Molecular Cell Biology*, 15(4):294–294, 2014. 2, 28

[29] S. Griffiths-Jones, A. Bateman, M. Marshall, A. Khanna, and S. R. Eddy. Rfam: an RNA family database. *Nucleic acids research*, 31(1):439–441, 2003. 50

[30] W. Gropp and E. L. Lusk. A taxonomy of programming models for symmetric multiprocessors and SMP clusters. In *Programming Models for Massively Parallel Computers, 1995*, pages 2–7. IEEE, 1995. 58

[31] S. A. Gupta S., Kececioglu J. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2(3):4590–472, 1995. 18, 19, 20, 81, 86, 110

[32] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997. 14, 15

[33] A. Harmanci, G. Sharma, and D. Mathews. Efficient pairwise RNA structure prediction using probabilistic alignment constraints in Dynalign. *BMC Bioinformatics*, 8, 2007. 45

[34] J. Harrow, A. Frankish, J. M. Gonzalez, E. Tapanari, and et al. GENCODE: the reference human genome annotation for The ENCODE Project. *Genome research*, 22(9):1760–1774, 2012. 2

[35] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. 20

[36] P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *ACM SIGART Bulletin*, (37):28–29, 1972. 20

[37] M. Hatem and W. Ruml. External memory best-first search for multiple sequence alignment. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence*, pages 409–416, 2013. 2, 3, 24, 84, 86, 88, 121

[38] J. Havgaard, E. Torarinsson, and J. Gorodkin. Fast pairwise structural RNA alignments by pruning of the dynamical programming matrix. *PLoS Computational Biology*, 3(10), 2007. 3, 5, 38, 39, 44, 47

[39] J. Havgaard, E. Torarinsson, and J. Gorodkin. Protocol s1. supplementary material for fast pairwise local structural RNA alignments by pruning of the dynamical programming matrix, 2007. Found at doi:10.1371/journal.pcbi.0030193.sd002 (97 KB PDF). 43

[40] J. H. Havgaard, R. B. Lyngsø, and J. Gorodkin. The foldalign web server for pairwise structural rna alignment and mutual motif search. *Nucleic acids research*, 33(suppl_2):W650–W653, 2005. 137

[41] J. H. Havgaard, R. B. Lyngsø, G. D. Stormo, and J. Gorodkin. Pairwise local structural alignment of rna sequences with sequence similarity less than 40%. *Bioinformatics*, 21(9):1815–1824, 2005. 43

[42] M. Helal, H. El-Gindy, L. Mullin, and B. Gaëta. Parallelizing optimal multiple sequence alignment by dynamic programming. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 669–674, 2008. 80

[43] M. Helal, L. Mullin, B. Gaëta, and H. El-Gindy. Multiple sequence alignment using massively parallel mathematics of arrays. In *International Conference on High Performance Computing, Networking and Communication Systems*, pages 120–127, 2007. 80

[44] M. Helal, L. Mullin, J. Potter, and V. Sintchenko. Search space reduction technique for distributed multiple sequence alignment. In *6th IFIP International Conference on Network and Parallel Computing*, pages 219–226. IEEE Computer Society, 2009. 80, 86

[45] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *National Academy of Sciences of the United States of America*, 89(22):10915–9, 1992. 10

[46] D. G. Higgins and P. M. Sharp. Clustal: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1):237–244, 1988. 26

[47] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977. 76

[48] M. P. Hoeppner, L. E. Barquist, and P. P. Gardner. An introduction to RNA databases. In J. Gorodkin and W. L. Ruzzo, editors, *In RNA Sequence, Structure, and Function: Computational and Bioinformatic Methods in Molecular Biology*, chapter 6, pages 107–123. Humana Press, 2014. 50

[49] I. Hofacker, S. Bernhart, and P. Stadler. Alignment of RNA base pairing probability matrices. *Bioinformatics*, 20(14):2222–2227, 2004. 38, 45, 46, 48

[50] I. Holmes. Accelerated probabilistic inference of RNA structure evolution. *BMC Bioinformatics*, 6:73, 2005. 45

[51] L. Huang, S. Ishibe-Murakami, D. Patel, and A. Serganov. Long-range pseudoknot interactions dictate the regulatory response in the tetrahydrofolate riboswitch. *Proceedings of the National Academy of Sciences*, 108(36):14801–6, 2011. 31

[52] C.-L. Hung, Y.-S. Lin, C.-Y. Lin, Y.-C. Chung, and Y.-F. Chung. CUDA ClustalW: An efficient parallel algorithm for progressive multiple sequence alignment on multi-gpus. *Computational biology and chemistry*, 58:62–68, 2015. 76, 85

[53] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill Science/Engineering/Math, 1st edition, 1998. 3, 57

[54] T. Ikeda and T. Imai. Fast A* algorithms for multiple sequence alignment. *Proceedings Genome Informatics Workshop V*, pages 90–99, 1994. 1, 23

[55] Intel. Intel Xeon Phi x100 Product Family, 2017. available at `https://ark.intel.com/products/series/92649/Intel-Xeon-Phi-x100-Product-Family` (accessed 18th October, 2017). 60

[56] Intel. Intel Xeon Phi x200 Product Family, 2017. available at `https://ark.intel.com/products/series/92650/Intel-Xeon-Phi-x200-Product-Family` (accessed 18th October, 2017). 60

[57] A. Jacob, J. Buhler, and R. D. Chamberlain. Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs. In *International Conference on Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008.*, pages 191–196. IEEE, 2008. xv, 4, 91, 92, 98

[58] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition*. Morgan Kaufmann Publishers Inc., 2nd edition, 2016. 3, 59

[59] Y. Jinnai and A. Fukunaga. Abstract zobrist hashing: An efficient work distribution method for parallel best-first search. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 717–723, 2016. 3, 87, 88, 121

[60] I. Kalvari, J. Argasinska, N. Quinones-Olvera, E. P. Nawrocki, E. Rivas, S. R. Eddy, A. Bateman, R. D. Finn, and A. I. Petrov. Rfam 13.0: shifting to a genome-centric resource for non-coding RNA families. *Nucleic Acids Research*, 2017. 50

[61] A. Kishimoto, A. Fukunaga, and A. Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artif. Intell.*, 195:222–248, 2013. 87

[62] R. J. Klein and S. R. Eddy. RSEARCH: Finding homologs of single structured RNA sequences. *BMC Bioinformatics*, 4(1):44, 2003. xvii, 51

[63] H. Kobayashi and H. Imai. Improvement of the A* algorithm for multiple sequence alignment. In *Proceedings of the 9th Workshop on Genome Informatics*, pages 120–130, 1998. 1, 23

[64] R. E. Korf. A divide and conquer bidirectional search: First results. In *IJCAI*, pages 1184–1191, 1999. 24

[65] R. E. Korf. Delayed duplicate detection: Extended abstract. In *IJCAI*, pages 1539–1541, 2003. 24

[66] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *Proceedings of the 19th National Conference on Artifical Intelligence*, pages 650–657, 2004. 24

[67] R. E. Korf. Linear-time disk-based implicit graph search. *Journal of the ACM*, 55(6), 2008. 25

[68] R. E. Korf. Linear-time disk-based implicit graph search. *Journal of the ACM (JACM)*, 55(6):26, 2008. 84

[69] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, and International Human Genome Sequencing Consortium, et al. Non-coding RNA human molecular genetics. *Nature*, 409(6822):860–921, 2001. 2

[70] C. Lee and R. Gutell. Diversity of base-pair conformations and their occurrence in rRNA structure and RNA structural motifs. *Journal of Molecular Biology*, 344(5):1225–49, 2004. 28

[71] G. Lei, Y. Dou, W. Wan, F. Xia, R. Li, M. Ma, and D. Zou. CPU-GPU hybrid accelerating the Zuker algorithm for RNA secondary structure prediction applications. *BMC genomics*, 13(1):S14, 2012. 4, 94, 98

[72] J. Li, S. Ranka, and S. Sahni. Multicore and GPU algorithms for Nussinov RNA folding. *BMC Bioinformatics*, 15(8):S1, 2014. xv, 4, 95, 98

[73] K. Li. ClustalW-MPI: ClustalW analysis using distributed and parallel computing. *Bioinformatics*, 19:1585–1586, 2003. 73, 85

[74] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *National Academy of Sciences of the United States of America*, 86(12):4412–4415, 1989. 19, 86

[75] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. GPU-ClustalW: using graphics hardware to accelerate multiple sequence alignment. In *HiPC*, pages 363–374. Springer, 2006. 75, 85

[76] Y. Liu, B. Schmidt, and D. L. Maskell. MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 121–128. IEEE Computer Society, 2009. 77, 85

[77] L. M. and R. K. The practical use of the A* algorithm for exact multiple sequence alignment. *Journal of Computational Biology*, 7:655–673, 2000. 1, 3, 24

[78] N. Markham and M. Zuker. DINAMelt web server for nucleic acid melting prediction. *Nucleic Acids Research*, 33:577–581, 2005. 94

[79] S. Masuno, T. Maruyama, Y. Yamaguchi, and A. Konagaya. An FPGA implementation of multiple sequence alignment based on Carrillo-Lipman method. In *2007 International Conference on Field Programmable Logic and Applications*, pages 489–492, 2007. xv, 18, 80, 81, 86, 106

[80] J. Mattick and I. Makunin. Non-coding rna. *Human Molecular Genetics*, 15(1):17–29, February 2006. 2, 29

[81] J. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29(6-7):1105–1119, 1990. 33

[82] M. A. McClure, T. K. Vasi, and W. M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Molecular Biology and Evolution*, 11(4):571–592, 1994. 20

[83] S. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Longman Ltd., 2001. 105

[84] S. Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14*. "O'Reilly Media, Inc.", 2014. 56

[85] B. Morgenstern, K. Frech, A. Dress, and T. Werner. DIALIGN: finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, 1998. 3, 26

[86] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966. 102

[87] D. W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 1st edition, 2001. 1, 8, 9, 14, 25, 26, 27, 28, 76

[88] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970. 10, 14

[89] R. Niewiadomski, J. N. Amaral, and R. Holte. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, AAAI'06, pages 1039–1044, 2006. 2, 3, 24, 83, 86, 88, 105, 106, 121

[90] C. Notredame. T-Coffee: a novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205–217, 2000. 26, 76

[91] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35(1):68–82, 1978. 32

[92] NVIDIA. PTX: Parallel Thread Execution, 2011. available at http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf (accessed 19th October, 2017). 62

[93] NVIDIA. NVIDIA Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012. available at http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (accessed 19th October, 2017). xv, 65, 66, 67

[94] NVIDIA. Tesla K40 GPU Active Accelerator, 2013. available at https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf accessed 24th October, 2017). 70

[95] NVIDIA. NVIDIA CUDA architecture introduction & overview, 2014. available at http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf (accessed 19th October, 2017). xv, 61

[96] NVIDIA. NVIDIA CUDA Programming Guide 6.5, 2014. xv, 61, 62, 63, 64, 77

[97] NVIDIA. NVIDIA Whitepaper NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made, 2014. available at https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (accessed 19th October, 2017). xv, 67, 68

[98] NVIDIA. NVIDIA GeForce GTX 1080 Gaming Perfected, 2016. available at https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (accessed 19th October, 2017). 69

[99] NVIDIA. Whitepaper, NVIDIA TESLA P100, 2016. available at https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (accessed 23th October, 2017). xv, xvii, 68, 69, 70, 71

[100] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. Maskell. Using reconfigurable hardware to accelerate multiple sequence alignment with clustalw. *Bioinformatics*, 21:3431–3432, 2005. 74, 85

[101] OpenCL. Website. http://www.khronos.org/opencl/ (accessed 20th October, 2017). 62

[102] The OpenMP® API specification for parallel programming, available at http://www.openmp.org (accessed 20th october, 2017). 54

[103] M. Palkowski and W. Bielecki. Parallel tiled Nussinov RNA folding loop nest generated using both dependence graph transitive closure and loop skewing. *BMC Bioinformatics*, 18(1):290, 2017. xv, 4, 96, 97, 98

[104] M. Palkowski and W. Bielecki. TRACO Website, 2017. available at http://traco.sourceforge.net (accessed 12th November, 2017). 97

[105] G. F. Pfister. *In search of clusters: the coming battle in lowly parallel computing*. Prentice-Hall, Inc., 1995. 90

[106] G. F. Pfister. *In search of clusters*. Prentice-Hall, Inc., 2nd edition, 1998. 53

[107] S. J. Reinert K. and W. T. Combining divide-and-conquer, the a-algorithm, and successive realignment approaches to speed multiple sequence alignment. In *German Conference on Bioinformatics*, pages 17–24, 1999. 1, 24

[108] G. Rizk and D. Lavenier. GPU accelerated RNA folding algorithm. In *ICCS (1)*, volume 5544 of *Lecture Notes in Computer Science*, pages 1004–1013. Springer, 2009. xv, 4, 93, 98

[109] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003. 1, 21, 22

[110] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. 3, 60, 61

[111] D. Sankoff. Simultaneous Solution of the RNA Folding, Alignment and Protosequence Problems. *SIAM Journal on Applied Mathematics*, 45(5):810–825, 1985. 3, 5, 38, 40

[112] S. Schrödl. An improved search algorithm for optimal multiple-sequence alignment. *Journal. Artif. Intell. Res. (JAIR)*, 23:587–623, 2005. 24

[113] T. Shibuya and H. Imai. New flexible approaches for multiple sequence alignment. In *Proceedings of the First Annual International Conference on Computational Molecular Biology*, RECOMB '97, pages 267–276, 1997. 1, 23

[114] D. Singh. Implementing FPGA design with the OpenCL standard. *Altera whitepaper*, 2011. 59

[115] R. Smith and T. Smith. Pattern-induced multi-sequence alignment (pima) algorithm employing secondary structure-dependent gap penalties for use in comparative protein modelling. *Protein Engineering*, 5, 1992. 27

[116] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. 12, 14

[117] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016. 60

[118] J. L. Spouge. Speeding up dynamic-programming algorithms for finding optimal lattice paths. *SIAM J. Applied Mathematics*, 49(5):1552–1566, 1989. 1, 3, 23

[119] A. Stark, P. Kheradpour, L. Parts, J. Brennecke, E. Hodges, G. J. Hannon, and M. Kellis. Systematic discovery and characterization of fly micrornas using 12 drosophila genomes. *Genome research*, 17(12):1865–1879, 2007. 94

[120] N. R. Sturtevant and J. Chen. External memory bidirectional search. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence, IJCAI 2016*, pages 676–682, 2016. 3, 87, 88, 121

[121] A. R. Subramanian, M. Kaufmann, and B. Morgenstern. DIALIGN-TX: greedy and progressive approaches for segment-based multiple sequence alignment. *Algorithms for Molecular Biology*, 3(1):6, 2008. 27

[122] D. Sundfeld, J. H. Havgaard, J. Gorodkin, and A. C. M. A. Melo. CUDA-Sankoff: Using GPU to accelerate the pairwise structural RNA alignment. In *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2017, 2017*, pages 295–302, 2017. 5, 6, 140, 167

[123] D. Sundfeld, J. H. Havgaard, A. C. M. A. Melo, and J. Gorodkin. Foldalign 2.5: multithreaded implementation for pairwise structural RNA alignment. *Bioinformatics*, 32(8):1238–1240, 2016. 5, 6, 123, 167

[124] D. Sundfeld and A. C. M. A. Melo. MSA-GPU: exact multiple sequence alignment using GPU. In J. C. Setubal and N. F. Almeida, editors, *BSB*, volume 8213 of *Lecture Notes in Computer Science*, pages 47–58. Springer, 2013. xv, 82, 86, 110

[125] D. Sundfeld, C. Razzolini, G. Teodoro, A. Boukerche, and A. C. M. A. Melo. PA-Star: A disk-assisted parallel A-Star strategy with locality-sensitive hash for multiple sequence alignment. *Journal of Parallel and Distributed Computing*, 112:154 – 165, 2018. xiv, 5, 6, 16, 100, 167

[126] D. Sundfeld, G. Teodoro, and A. C. M. A. Melo. Parallel A-Star multiple sequence alignment with locality-sensitive hash functions. In *9th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015, 2015*, pages 342–347, 2015. 5, 6, 100, 167

[127] A. S. Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009.

[128] W. Taylor. A flexible method to align large numbers of biological sequences. *Journal of Molecular Evolution*, 28:161–169, 1988. 27

[129] J. D. Thompson, D. G. Higgins, and T. J. Gibson. Clustal W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–4680, 1994. 3, 26

[130] J. D. Thompson, F. Plewniak, and O. Poch. BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics*, 15(1):87–88, 1999. 84, 110

[131] E. Torarinsson, J. Havgaard, and J. Gorodkin. Multiple structural alignment and clustering of RNA sequences. *Bioinformatics*, 23(8):926–932, 2007. 39, 47, 48

[132] A. Vrenios. *Linux Cluster Architecture*. Sams Pub Co., 2002. 57

[133] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994. 1, 14

[134] S. Will, K. Reiche, I. L. Hofacker, P. F. Stadler, and R. Backofen. Inferring noncoding RNA families and classes by means of genome-scale structure-based clustering. *PLoS Computational Biology*, 3(4), 2007. 3, 38, 45, 46

[135] W. Wolf. *FPGA-Based System Design*. Prentice Hall PTR, 2004. 3, 58

[136] M. Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986. 97

[137] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *AAAI/IAAI*, pages 923–929, 2000. 24, 25

[138] M. Yusupov, G. Yusupova, A. Baucom, K. Lieberman, T. Earnest, J. Cate, and H. Noller. Crystal structure of the ribosome at 5.5 Å resolution. *Science*, 292(5518):883–896, 2001. 2, 28

[139] R. Zhou and E. A. Hansen. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *In Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 427–434, 2003. 24

[140] Y. Zhou and J. Zeng. Massively parallel A* search on a GPU. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, pages 1248–1255, 2015. 3, 86, 88, 121

[141] M. Ziv-Ukelson, I. Gat-Viks, Y. Wexler, and R. Shamir. A faster algorithm for RNA co-folding. In *Proceedings of the 8th International Workshop on Algorithms in Bioinformatics*, pages 174–185, 2008. xv, 37, 39, 41

[142] Q. Zou, Q. Hu, M. Guo, and G. Wang. HAlign: Fast multiple similar DNA/RNA sequence alignment based on the centre star strategy. *Bioinformatics*, 31(15):2475–2481, 2015. 79

Anexo A

Artigos Decorrentes desta Tese

A.1 Artigos Publicados em Periódicos Internacionais

- [123] D. Sundfeld, J. H. Havgaard, A. C. M. A. Melo, and J. Gorodkin. Foldalign 2.5: multithreaded implementation for pairwise structural RNA alignment. *Bioinformatics*, 32(8):1238–1240, 2016. (Journal Qualis A1, JCR=7.307)
- [125] D. Sundfeld, C. Razzolini, G. Teodoro, A. Boukerche, and A. C. M. A. Melo. PA-Star: A disk-assisted parallel A-Star strategy with locality-sensitive hash for multiple sequence alignment. *Journal of Parallel and Distributed Computing*, 112:154–165, 2018. (Journal Qualis A2, JCR=1.93)

A.2 Artigos Publicados em Conferências Internacionais

- [126] D. Sundfeld, G. Teodoro, and A. C. M. A. Melo. Parallel A-Star multiple sequence alignment with locality-sensitive hash functions. In *9th International Conference on Complex, Intelligent, and Software Intensive Systems, Workshop EPAMUS, 2015,* pages 342–347, 2015. (Conferência Qualis B5)
- [122] D. Sundfeld, J. H. Havgaard, J. Gorodkin, and A. C. M. A. Melo. CUDA-Sankoff: Using GPU to accelerate the pairwise structural RNA alignment. In *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2017, 2017*, pages 295–302, 2017. (Conferência Qualis A2)

A.3 Primeira Página dos Artigos em Ordem de Publicação

2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems

Parallel A-Star Multiple Sequence Alignment with Locality-Sensitive Hash Functions

Daniel Sundfeld, George Teodoro and Alba Cristina M. A. de Melo
Department of Computer Science, University of Brasilia
Brasília, DF - Brazil
Email: sund@unb.br, teodoro@cic.unb.br, albamm@cic.unb.br

***Abstract*—In this paper, we propose and evaluate a parallel solution for the exact Multiple Sequence Alignment problem based on the A-Star algorithm. In our parallel solution, we use a multi-index data structure, templates and a locality-sensitive hash function. The results were collected in two machines (4 cores and 32 cores), with real and synthetic sequence sets ranging from 3 to 14 sequences. We show that our parallel solution executes 2.89× and 4.77× faster than a state-of-the-art parallel MSA tool, with a proportional increase in memory usage, when comparing 2 hard instances of the benchmark Balibase reference set 1.**

I. Introduction

Bioinformatics is an interdisciplinary field that involves computer science, biology, mathematics and statistics [1]. One of its main goals is to analyze biological sequence data and genome content in order to obtain the function/structure of the sequences as well as evolutionary information.

Sequence comparison is one of the most basic operations in Bioinformatics and it is used to visualize the similarity and differences between the sequences. A sequence can be compared to another sequence (Pairwise Comparison) or to a set of sequences (Multiple Sequence Alignment).

In a Multiple Sequence Alignment (MSA), similar characters among a set of k sequences ($k \geq 3$) are aligned together. Multiple Sequence Alignments are often used as a building block to solve important and complex problems in Molecular Biology, such as the identification of conserved motifs, definition of phylogenetic relationships and 3D homology modeling, among others. In all these cases, the quality of the solutions relies heavily on the quality of the underlying MSA.

The MSA problem has been proven as NP-Hard [2]. Carrillo-Lipman [3] made an important contribution by showing it is not necessary to explore the whole search space to obtain the optimal alignment, defining an upper and a lower bound. Spouge [4] demonstrated that the lower bound is an admissible heuristic for the A-Star (A*) algorithm [5].

Many efforts have been made to reduce the search space of the A-Star algorithm [4], [6]-[8]. A few efforts were also made to implement the exact Multiple Sequence Alignment in parallel versions [9], [10]. Nevertheless, obtaining an exact MSA solution with good performance is still a challenging problem.

In this paper, we propose and evaluate a parallel solution for the exact MSA problem that reduces the search-space using the A-Star (A*) algorithm. In our solution, we divide the search-space using hash functions with locality preserving characteristic, aiming to provide a better use of caches and to reduce the overhead of threads communication. The results were initially obtained in a 4-cores machine with 8 GB of RAM memory, then, to solve more complex problems, a 32-cores machine with 1TB of RAM was used. The results obtained by comparing real and synthetic sets of sequences on these machines show that we are able to drastically reduce the execution time. We show that our parallel approach is able to outperform one of the state-of-the art solutions [10], with a speedup of $2.89\times$ and $4.77\times$ for the Balibase glg and 2ack instances.

The rest of this paper is organized as follows. We present an overview of the MSA problem in Section II. Section III discusses related work in the area of Exact MSA. In Section IV, we present the design of our parallel strategy and the experimental results are shown in Section V. Finally, we conclude the paper in Section VI.

II. Multiple Sequence Alignment

To compare two sequences (pairwise comparison), it is necessary to find the best alignment between them, which amounts to place one sequence above the other making clear the correspondence between similar characters [1].

A global Multiple Sequence Alignment (MSA) of $k \geq 3$ sequences $S = S_1, S_2, ..., S_k$ is obtained in such a way that spaces (gaps) are inserted into each of the k sequences so that the resulting sequences have the same length l. Then, the sequences are arranged in k rows of l columns each [1]. Figure 1 shows an example of an MSA of 3 DNA sequences.

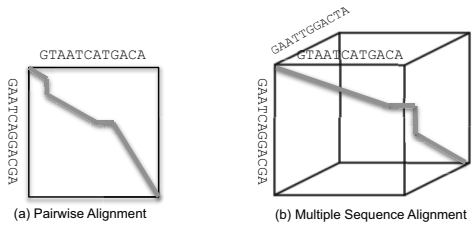


Fig. 1: Pairwise alignment and MSA with 3 sequences. The gray line represents one possible alignment.

Usually, MSAs are scored with the Sum-of-Pairs (SP) function and the exact SP MSA problem is known to be NP-hard [2]. In SP, every pair of bases is scored with the pairwise scoring function and the final score is the addition of all these values. For instance, considering that the punctuation for matches (similar characters), mismatches (different characters) and gaps are 0, $+1$ and $+1$, respectively, the score generated by pairwise comparison of sequences S_1 and S_2 is 5 (Figure 2). The SP score of the MSA in this example is 16.

The optimal MSA among n sequences can be calculated by extending the exact dynamic programming (DP) algorithm for

978-1-4799-8870-9/15 $31.00 © 2015 IEEE
DOI 10.1109/CISIS.2015.50

Bioinformatics, 32(8), 2016, 1238–1240
doi: 10.1093/bioinformatics/btv748
Advance Access Publication Date: 24 December 2015
Applications Note

OXFORD

Structural bioinformatics

Foldalign 2.5: multithreaded implementation for pairwise structural RNA alignment

Daniel Sundfeld[1,2], Jakob H. Havgaard[1], Alba C. M. A. de Melo[2] and Jan Gorodkin[1,*]

[1]Center for Non-Coding RNA in Technology and Health, IKVH, University of Copenhagen, Frederiksberg, Denmark and [2]Department of Computer Science, University of Brasilia, Brasília, DF, Brazil

*To whom correspondence should be addressed.
Associate Editor: Ivo Hofacker

Received on 23 July 2015; revised on 14 December 2015; accepted on 16 December 2015

Abstract

Motivation: Structured RNAs can be hard to search for as they often are not well conserved in their primary structure and are local in their genomic or transcriptomic context. Thus, the need for tools which in particular can make local structural alignments of RNAs is only increasing.
Results: To meet the demand for both large-scale screens and hands on analysis through web servers, we present a new multithreaded version of Foldalign. We substantially improve execution time while maintaining all previous functionalities, including carrying out local structural alignments of sequences with low similarity. Furthermore, the improvements allow for comparing longer RNAs and increasing the sequence length. For example, lengths in the range 2000–6000 nucleotides improve execution up to a factor of five.
Availability and implementation: The Foldalign software and the web server are available at http://rth.dk/resources/foldalign
Contact: gorodkin@rth.dk
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Recent research points towards an increasing awareness of structured RNAs in genomic and transcriptomic sequences (Gorodkin *et al.*, 2010; Westhof and Romby, 2010). However, the tools needed for structural analysis, such as pairwise local structural RNA alignments, are not yet fully developed.

Foldalign (Havgaard *et al.*, 2007) is a tool that explicitly carries out local pairwise structural alignment of RNA sequences based on the Sankoff algorithm (Sankoff, 1985). Even though tools like CMfinder can carry out local RNA structure alignment on multiple sequences (Yao *et al.*, 2006), the pairwise problem is still of key interest. A range of other methods for RNA structural alignments focusing more on the global alignment (Dowell and Eddy, 2006; Knudsen and Hein, 2003) have been proposed, but only a few efforts were made to parallelize these methods (Fu *et al.*, 2014; Sukosd *et al.*, 2011).

The relevancy of a parallel version of the pairwise Sankoff algorithm is underpinned by its time complexity of $O(L^6)$, where L is the sequence length. This makes it prohibitive for long sequences, but Foldalign uses several heuristics: a maximum length of the alignment, λ, and a maximum difference, δ between any two subsequences being aligned. This reduces the time complexity to $O(L^2\lambda^2\delta^2)$. However, runtime and memory is further substantially improved using several other heuristics like limiting the multiloop calculation and pruning of the alignment score, for details see Havgaard *et al.* (2007). All these heuristics can be used in a parallel version of the algorithm, which we address here, providing new opportunities for both large-scale analysis as well as case-based analyses through a web interface.

2 Implementation and results

The core Foldalign algorithm has six nested loops (Supplementary Section 1) which are subject for parallelization. The first and second

2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing

CUDA-Sankoff: using GPU to accelerate the pairwise structural RNA alignment

Daniel Sundfeld*, Jakob H. Havgaard†, Jan Gorodkin†, Alba C. M. A. de Melo*
*Department of Computer Science,
University of Brasilia, Brasilia, DF, Brazil
Email: {sund,alves}@unb.br
†Center for non-coding RNA in Technology and Health,
IKVH, University of Copenhagen, Denmark
Email: {hull,gorodkin}@rth.dk

***Abstract*—In this paper, we propose and evaluate CUDA-Sankoff, a solution to the RNA structural alignment problem based on the Sankoff algorithm in Graphics Processing Units (GPUs). To our knowledge, this is the first time the Sankoff algorithm is implemented in GPU. In our solution, we show how to linearize the Sankoff 4-dimensional dynamic programming (4D DP) matrix and we propose a two-level wavefront approach to exploit the parallelism. The results were obtained with two different NVidia GPUs, comparing sets of real RNA sequences with lengths from 46 to 281 nucleotides. We show that our GPU approach is up to 24 times faster than a 16-core CPU solution in the 281 nucleotide Sankoff execution.**

I. Introduction

Bioinformatics is an interdisciplinary area that involves computer science, statistics, mathematics and biology, aiming to create algorithms and tools to help biologists in their data analysis [1]. Sequence comparison is one of the most basic operations in Bioinformatics and its goal is to expose the evolutionary relationship among two or more sequences by computing a similarity score and an alignment. A comparison operation can take into account the biological sequences themselves (1D comparison) or their structure (2D comparison).

Structural comparisons are usually done for Ribonucleic acid (RNA) sequences, which form hydrogen bonds between cytosine (C) and guanine (G), between adenine (A) and uracil (U) and between G and U, folding over themselves thus generating a 2D structure. Nussinov et al. [2] proposed an algorithm based on dynamic programming to predict the 2D structure of a given RNA sequence. This algorithm was further extended by Sankoff [3] for more than one sequence, giving as output the 2D structure of a set of sequences, folding and aligning the sequences simultaneously. Sankoff's algorithm runs in $O(L^{3N})$ time and uses $O(L^{2N})$ memory, for N sequences of length L and provides the optimal solution.

Due to its high computing cost and huge memory requirements, Sankoff's algorithm is considered unfeasible. For this reason, many practical Bioinformatics tools such as Foldalign [4] and Dynalign [5] implement a constrained version of Sankoff's, limiting the size of the alignment and/or the alignment characteristics. With this, the solution can be retrieved in much less time but it is no longer optimal.

Graphics Processing Units (GPUs) provide impressive parallelism, with more than 1,000 cores working in parallel, and can be a good alternative to implement the Sankoff algorithm. However, this algorithm presents complex data dependency patterns and thus its parallelization is an arduous task. To our knowledge, there is no work in the literature that implements Sankoff in GPU.

Most of the current implementations of the Sankoff algorithm include a range of heuristics to reduce time and memory complexity [4], [5]. These heuristic methods do not provide the optimal result and, thus, are beyond the scope of this paper, where we focus on how GPUs can be exploited for RNA optimal structural alignments. Hence, in this paper, we propose and evaluate CUDA-Sankoff, a GPU solution to implement an unconstrained version of the Sankoff algorithm. In our solution, we use GPU's parallelism to calculate simultaneously several cells in Sankoff's 4D dynamic programming (DP) matrix. The 4D DP matrix is linearized in our solution in order to accelerate memory accesses. In addition, we calculate the linearized DP matrix with a two-level wavefront that allows to attain very good parallelism.

The results obtained with real RNA sequences using separately two GPUs and one CPU (1 core and 16 cores) show that the GPUs can attain impressive speedups over the CPU. We show that, for sequences with lengths from 46 to 281, our GPU approach is able to outperform a single-thread CPU solution, achieving a speedup of 109x. When compared to an OpenMP-based 16-core solution, CUDA-Sankoff is able to attain a speedup of 24x. In this case, the execution time was reduced from 28 hours and 39 minutes to 1 hour and 11 minutes.

The remainder of this paper is organized as follows. We present an overview of the Structural RNA Alignment problem in Section II. Section III discusses related work in the area of Structural RNA Alignment. In Section IV, the design of CUDA-Sankoff is presented and the results are shown in Section V. Finally, we conclude the paper and give the future work directions in Section VI.

2377-5750/17 $31.00 © 2017 IEEE
DOI 10.1109/PDP.2017.15

J. Parallel Distrib. Comput. 112 (2018) 154–165

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

PA-Star: A disk-assisted parallel A-Star strategy with locality-sensitive hash for multiple sequence alignment

Daniel Sundfeld [a], Caina Razzolini [a], George Teodoro [a], Azzedine Boukerche [b], Alba Cristina Magalhaes Alves de Melo [a,*]

[a] Department of Computer Science, University of Brasilia (UnB), Brazil

[b] School of Information Technology and Engineering (SITE), University of Ottawa, Canada

HIGHLIGHTS

- An A-Star based algorithm that retrieves optimal multiple sequence alignments.
- Locality-sensitive hash functions to assign work to cores.
- Disk-assisted strategy which augments the amount of memory.
- Better performance than state-of-the-art.

ARTICLE INFO

Article history:
Received 21 August 2016
Received in revised form
3 February 2017
Accepted 29 April 2017
Available online 26 May 2017

Keywords:
Multiple sequence alignment
Locality-sensitive hash
A-Star
Parallel algorithms

ABSTRACT

Multiple Sequence Alignment (MSA) is a basic operation in Bioinformatics, and is used to highlight the similarities among a set of sequences. The MSA problem was proven NP-Hard, thus requiring a high amount of memory and computing power. This problem can be modeled as a search for the path with minimum cost in a graph, and the A-Star algorithm has been adapted to solve it sequentially and in parallel. The design of a parallel version for MSA with A-Star is subject to challenges such as irregular dependency pattern and substantial memory requirements. In this paper, we propose PA-Star, a locality-sensitive multithreaded strategy based on A-Star, which computes optimal MSAs using both RAM and disk to store nodes. The experimental results obtained in 3 different machines show that the optimizations used in PA-Star can achieve an acceleration of 1.88× in the serial execution, and the parallel execution can attain an acceleration of 5.52× with 8 cores. We also show that PA-Star outperforms a state-of-the-art MSA tool based on A-Star, executing up to 4.77× faster. Finally, we show that our disk-assisted strategy is able to retrieve the optimal alignment when other tools fail.

1. Introduction

Bioinformatics is an interdisciplinary field that involves computer science, biology, mathematics and statistics [4]. One of its main goals is to analyze biological sequence and genome data in order to obtain the function/structure of the sequences, as well as evolutionary information.

Multiple Sequence Alignment (MSA) is a basic operation in Bioinformatics, in which similar characters among a set of n biological sequences ($n \geq 3$) are aligned together to highlight the similarity among the sequences. MSAs are often used as a building block for solving important and complex problems, such as the definition of phylogenetic relationships and 3D structure prediction, among others. In all these cases, the quality of the solutions relies heavily on the quality of the underlying MSAs.

The MSA problem has been proven NP-Hard [24] and, for this reason, obtaining the optimal solution requires high computing power and a large amount of memory. There are two main strategies for reducing the search space of the optimal MSA problem. The first was proposed by Carrillo–Lipman [2] and defines lower and upper bounds in the search space composed of an n-dimensional dynamic programming matrix, in which n is the number of sequences. The second strategy transforms the MSA problem in the problem of searching the path with minimum cost in a graph and applying the A-Star (A*) algorithm [5] to solve it. In this paper, we employ the A-Star based strategies.

* Corresponding author.
E-mail addresses: sund@unb.br (D. Sundfeld), cfbrazzolini@gmail.com (C. Razzolini), teodoro@cic.unb.br (G. Teodoro), boukerch@site.uottawa.ca (A. Boukerche), alves@unb.br (A.C.M.A. de Melo).

http://dx.doi.org/10.1016/j.jpdc.2017.04.014
0743-7315/