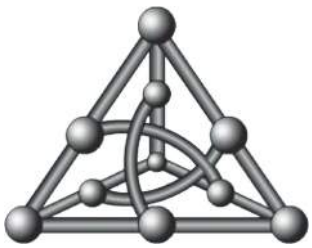
Algoritmos Paralelos para o Alinhamento de Sequências Genômicas

Pedro Henrique Neves da Silva

Orientação: Prof. Dr. Marco Aurélio Stefanes

Área de Concentração: Algoritmos Paralelos

Dissertação de Mestrado.



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
Campo Grande, 24 de abril de 2014

Algoritmos Paralelos para o Alinhamento de Sequências Genômicas

Pedro Henrique Neves da Silva

DISSERTAÇÃO APRESENTADA À FACULDADE DE COMPUTAÇÃO DA UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO

Este exemplar corresponde à redação final da dissertação de mestrado devidamente corrigida e defendida por Pedro Henrique Neves da Silva e aprovada pela Banca Examinadora.

Banca Examinadora:

- Prof. Dr. Marco Aurelio Stefanes (Orientador) (FACOM / UFMS)
- Prof. Dr. Francisco Eloi Soares de Araujo (FACOM / UFMS)
- Prof. Dr. Luiz Carlos da Silva Rozante (UFABC)

Dedicatória

Dedico este trabalho à mulher que mudou a minha vida, minha linda esposa Karolina.

Agradecimentos

Agradeço primeiramente à Deus pela força e capacitação para concluir este mestrado. Minha sincera gratidão, ao Senhor digno de toda honra e toda glória, por tudo que tem feito em minha vida e por tudo que ainda há de fazer.

Agradeço aos meus pais, Regina e Marcos, por estarem comigo me suportando.

À minha esposa, Karolina, que me incentivou e se preocupou com o meu objetivo. Agradeço, também, aos meus sobrinhos, Davi e Heloísa que me alegraram em momentos de preocupação.

Por fim, agradeço à CAPES pelo apoio financeiro durante meu mestrado.

Resumo

No estudo da evolução dos organismos, ou das funções biológicas das moléculas, é comum a comparação entre diferentes organismos, ou moléculas, onde, em geral, essas moléculas são DNA, RNA ou proteínas, que são facilmente representadas por sequências de caracteres. A análise dessas várias sequências é um problema que necessita de muito tempo para ser realizada. Visando diminuir esse tempo são desenvolvidos métodos utilizando programação paralela com granulosidade híbrida, sendo essa paralelização necessária para tratar várias sequências com mais de 1000 caracteres.

Neste trabalho estudamos o alinhamento de várias sequências e implementamos um algoritmo paralelo para este problema e comparamos o desempenho com o algoritmo sequencial utilizado pelo *ClustalW*, obtendo *speedups* que variam entre 61 e 8200, e com o algoritmo paralelo utilizado pelo *ClustalW-MPI*, obtendo *speedups* que variam entre 44 e 280, quando temos muitas sequências de tamanho pequeno e quando temos um número considerável de sequências de tamanho grande, respectivamente, em ambas as comparações.

Palavras-chave: Alinhamento de Várias Sequências, ClustalW, MPI, CUDA.

Abstract

In the study of evolution and biological functions of molecules is common to compare different organisms or molecules, where, in general, these molecules are DNA, RNA or proteins that are easily represented by sequences of characters. The analysis of these sequences, either in pairs or in multiple sequences, is a problem that needs much time to be performed. And, aiming to reduce that time, parallel programming methods are developed, using hybrid granularity, and this parallelization is required to treat sequences in pratical scales.

We have studied the multiple sequence alignment and implemented a parallel algorithm for this problem and we have compared the performance with the sequential algorithm used by the *ClustalW*, obtaining speedups between 61 and 8200, and with the parallel algorithm *ClustalW-MPI*, obtaining speedups between 44 and 280, when we have many small sequences and when we have few sequences with big size.

Keywords: Multiple Sequence Alignment, ClustalW, MPI, CUDA.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

No estudo da evolução dos organismos, ou das funções biológicas das moléculas, é comum a comparação entre diferentes organismos, ou moléculas, onde, em geral, essas moléculas são DNA, RNA ou proteínas, que são facilmente representadas por sequências de caracteres. O Alinhamento de Sequências é o problema de alinhar sequências de caracteres representando DNA, RNA ou aminoácidos (ou fragmento destes) de forma a emparelhar tantos caracteres quanto possíveis de cada sequência. Dado um esquema para avaliar pares de caracteres e possíveis *gaps*, que são espaços inseridos nas sequências, o problema consiste em atribuir *gaps* em cada sequência de forma a maximizar ou minimizar a pontuação geral do alinhamento.

O principal propósito do Alinhamento de Sequências é inferir homologia entre sequências, ou seja, verificar se as sequências possuem um ancestral comum [dS08]. Entre outras aplicações para o problema podemos citar: encontrar padrões de diagnósticos para caracterizar famílias de proteínas, predizer estruturas secundárias de RNA ou proteínas, detectar similaridade entre novas sequências e famílias já conhecidas de sequências e ajudar na análise evolucionária [THG94]. Quando temos duas sequências a serem alinhadas temos soluções eficientes baseadas no paradigma de programação dinâmica. Entretanto, o alinhamento de várias sequências é um problema NP-difícil com algoritmos de aproximação sendo usados nesta tarefa [DE06]. Ou seja, buscamos uma solução próxima da ótima, não exatamente a ótima, com algoritmos mais rápidos, geralmente aproveitando-se do fato de que sequências homólogas são relacionadas evolutivamente.

Dado que sequências genômicas podem ter milhares de bases nitrogenadas e aminoácidos e que a quantidade de sequências também pode chegar na

casa de milhares, os algoritmos de aproximação ainda exigem um dispêndio de tempo considerável, dias ou semanas dependendo do caso, para obtermos um alinhamento perto do ótimo. Neste sentido, uma maneira natural de reduzir o tempo de busca da solução é utilizar versões paralelas dos algoritmos. Essas soluções podem ser comparadas pela granulosidade, arquitetura alvo e modelo de paralelização utilizado [LSM09]. Soluções com granulosidade grossa são desenvolvidas para arquiteturas com memória distribuída, *cluster* e multiprocessadores simétricos (SMP), enquanto que, soluções com granulosidade fina, focam em processadores com múltiplos núcleos e aceleradores, tais como *threads*, FPGAs e GPUs. Além desses dois tipos, ainda temos as soluções com granulosidade híbrida que são desenvolvidas usando métodos da granulosidade grossa para a comunicação entre GPUs, ou entre *threads*.

O objetivo desta dissertação é apresentar uma solução que paraleliza todos os estágios do algoritmo *ClustalW-MPI* com granulosidade híbrida, o que não temos na literatura, para o alinhamento global de várias sequências utilizando um *cluster* de GPUs.

Para tal, devemos revisar alguns conceitos e algoritmos que serão utilizados. Na Seção 1.1, temos as definições para alfabetos, símbolos, sequências e alinhamentos que utilizaremos no texto. O alinhamento de pares de sequências e a sua solução por programação dinâmica é apresentado no Capítulo 2 e a sua versão para várias sequências é apresentado no Capítulo 3, onde temos uma outra versão que utiliza heurística para resolver o problema. No Capítulo 4, temos as definições dos modelos computacionais BSP e CGM, da interface MPI e da arquitetura CUDA. O Capítulo 5 mostra duas implementações paralelas de um mesmo algoritmo para alinhamento de várias sequências e o Capítulo 6 mostra a nossa implementação híbrida utilizando MPI e CUDA para o problema do alinhamento de várias sequências. No Capítulo 7 temos os resultados obtidos pela implementação e no Capítulo 8 apresentamos a conclusão da dissertação.

1.1 Definições

Para definirmos formalmente alinhamentos de pares de sequências, ou de várias sequências, temos que fazer algumas definições para estabelecer a linguagem utilizada e fixar notações, utilizando como base as referências: [SM97, dB03, Lou10, Sil03, dS08].

1.1.1 Alfabetos, Símbolos e Sequências

De uma forma geral, sequência é uma sucessão ordenada de caracteres, ou símbolos, de um alfabeto, que é conjunto finito e não vazio de símbolos. Quando falamos de alinhamentos de sequências genômicas, o *alfabeto* Σ é correspondente às bases nitrogenadas presentes em sequências de DNA, ou seja, $\Sigma = \{A,\ C,\ G,\ T\}$. Um elemento $\sigma \in \Sigma$ é chamado de *símbolo*, ou *caractere*, e uma *sequência* s sobre o alfabeto Σ é uma sequência $s = (\sigma_1,\ \sigma_2,\ \ldots,\ \sigma_n) \in \Sigma^n$, onde $n \geq 0$ é um inteiro. Por conveniência, denotaremos s por $\sigma_1,\ \sigma_2,\ \ldots,\ \sigma_n$. Se $s = \sigma_1,\ \sigma_2,\ \ldots,\ \sigma_n$, denotamos o *comprimento* de s por $|s| = n$.

Uma *subsequência* s' de s é uma sequência $s' = \sigma_{i1},\ \sigma_{i2},\ \ldots,\ \sigma_{im}$ em que $\{i_1,\ i_2,\ \ldots,\ i_m\} \subseteq \{1,\ 2,\ \ldots,\ n\}$ e $i_1 < i_2 < \ldots < i_m$. Nesse caso, também dizemos que s é uma *super sequência* de s'. É importante observar que, enquanto um segmento de s é uma série de caracteres consecutivos de s, uma subsequência pode ser constituída de caracteres que não necessariamente estejam consecutivos em s.

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

1.1.2 Alinhamentos

Sejam $A = a_1 a_2 \ldots a_n$ e $B = b_1 b_2 \ldots b_m$ duas sequências sobre o mesmo alfabeto Σ, com $|A| = n$ e $|B| = m$. Para alinhar as duas sequências inserimos espaços nas duas sequências de tal forma que, ao final, possuam o mesmo comprimento. Um alinhamento entre A e B é um emparelhamento de símbolos $a \subset A$ e $b \subset B$ sujeito à restrição de que se linhas são traçadas entre os símbolos emparelhados, as linhas não podem se interceptar. Um alinhamento mostra a similaridade entre duas sequências onde a similaridade é uma medida do quanto as sequências são parecidas, ou seja, refere-se à porcentagem de bases nitrogenadas idênticas. O que buscamos é um alinhamento com similaridade máxima. Formalmente, sejam s *e* t sequências sobre um alfabeto $\Sigma' = \Sigma \cup \{\sqcup\}$, com $|s| = n$ e $|t| = m$. Um *alinhamento* A de s *e* t é uma matriz A de dimensões $2 \times n$, onde $n \geq \max\{n_1, n_2\}$, com entradas de Σ' tal que a linha A_1 do alinhamento A contém a sequência s com possíveis espaços inseridos entre os caracteres de s e a linha A_2 contém a sequência t com possíveis espaços inseridos entre os caracteres de t. Dizemos

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

O alinhamento de várias sequências é uma generalização do alinhamento de pares de sequência, onde temos várias sequências que devem ser alinhadas por inserção de espaços nas sequências até que todas tenham o mesmo tamanho. Formalmente, sejam k um inteiro positivo e $s_1,\ s_2,\ ...,\ s_k$ sequências sobre um alfabeto $\Sigma' = \Sigma \cup \{\sqcup\}$, com $|s_i| = n_i$, para $i = 1,\ 2,\ ...,\ k$. Um *alinhamento* A de $s_1, s_2, ..., s_k$ é uma matriz $A = (A_{ij})$ de dimensões $k \times n$, onde $n \geq \max\{n_i\}$, onde $1 \geq i \geq k$,com entradas de Σ' tal que, para cada i, existe um conjunto $J_i = \{j_1, j_2, \ldots, j_{n_i}\} \subseteq \{1, 2, \ldots, n\}$, com $j_1 < j_2 < ... < j_{n_i}$, tal que $A_{ij_1}, A_{ij_2}, \ldots, A_{ij_{n_i}} = s_i$ e para todo $j \in \{1, 2, \ldots, n\} - J_i$, temos $A_{ij} = \sqcup$.

1.1.3 Pontuação

Existem várias formas para se avaliar o alinhamento entre duas sequências. O método mais comum, simples e que será utilizado neste texto é denominado de distância de edição, que avalia o custo de fazer inserções e remoções nas sequências, e será melhor explicada na Seção 1.1.3. Uma alternativa de avaliar duas sequências é através da medida de similaridade. Nessa abordagem, o interesse está no melhor alinhamento entre as sequências e a pontuação desse método mostra o quão parecidas as sequências são.

Matrizes de Pontuação

Para determinarmos a pontuação de um alinhamento A de duas sequências s e t sobre um alfabeto Σ, usamos uma função auxiliar $p : \Sigma' \times \Sigma' \to \mathbb{Q}$ que a cada par de símbolos (σ, ρ) de Σ' associa uma pontuação $p(\sigma, \rho)$ para alinharmos σ e ρ. Uma tal função é denominada *função de pontuação* e, geralmente, é representada por uma matriz.

Usando a matriz de pontuação p, definimos a pontuação ou custo $c(A)$ do alinhamento A entre s e t como

$$c(A) = \sum_{j=1}^{n} p(s[j], t[j]).$$

Essa forma de atribuir pontuação a alinhamentos faz sentido do ponto de vista biológico, pois *gaps* podem ser interpretados como inserções, ou

deleções, de caracteres de uma das sequências. Além disso, quando alinhamos duas sequências que compartilham um ancestral comum, a não similaridade entre elas em uma posição pode ser interpretada como uma mutação ocorrida em uma delas.

Métricas

Das classes de matrizes de pontuação, a que utilizaremos em nossos estudos é a classe das métricas, que é definida como uma função $d : M \times M \to \mathbb{Q}_{\geq 0}$ definida sobre um conjunto não-vazio qualquer M que satisfaça, para todo $x, y, z \in M$

(i) $d(x, y) \geq 0$;

(ii) $d(x, y) = 0 \iff x = y$;

(iii) $d(x, y) = d(y, x)$ (simetria);

(iv) $d(x, y) \leq d(x, z) + d(z, y)$ (desigualdade triangular).

Logo, dada uma métrica, representada por uma matriz de pontuação p, a *distância* $d(s, t)$ entre as sequências s e t é

$$d(s, t) = \min_{A \in \mathcal{A}_{s,t}} \{p(A)\},$$

onde $\mathcal{A}_{s,t}$ é o conjunto de todos os alinhamentos de s e t.

Um alinhamento entre duas sequências é dito *ótimo* se sua pontuação é mínima.

Distância de Edição

Dentre as métricas, uma que merece destaque é aquela derivada a partir da matriz de pontuação zero-ou-um, isto é, quando $c(\sigma, \rho) = 0$ para $\sigma = \rho$ e $c(\sigma, \rho) = 1$ para $\sigma \neq \rho$. Nesse caso, a distância $d(s, t)$, derivada de c, é chamada de *distância de edição*.

A distância de edição entre sequências também pode ser definida como o número de operações de edição necessárias para transformar uma sequência em outra. Para isso temos as operações de edição de *substituição*, *remoção* e *inserção* de caracteres, que possuem interpretação biológica durante o processo evolutivo de sequências. No caso geral em que tais operações de edição

não têm custo uniforme, utilizamos a mesma definição da matriz de pontuação zero-ou-um com a diferença que devemos minimizar o custo da sequência de transformações.

Nesse caso geral, quando o custo das operações satisfaz à desigualdade triangular, vemos que essa definição coincide com a definição de que a distância é a pontuação do alinhamento de menor custo, o que pode ser justificado pelo fato de que um alinhamento entre duas sequências codifica uma sequência de operações de transformação entre as sequências alinhadas.

Capítulo 2

Alinhamento de Pares de Sequências

Neste trabalho, estamos interessados em alinhamentos globais que exponham da melhor forma possível a estrutura das sequências consideradas, evidenciando suas similaridades e diferenças e permitindo, assim, uma fácil comparação. Para tanto, precisamos atribuir um "conceito de qualidade" ou "pontuação" para cada alinhamento, de tal forma que possamos escolher o melhor alinhamento. Na Figura 2.1, mostramos dois possíveis alinhamentos para as sequências `TAGGTCA` e `TAGCTA`.

```
T A G G T C A ␣ ␣ ␣ ␣ ␣ ␣
␣ ␣ ␣ ␣ ␣ ␣ ␣ T A G C T A

T A G G T C A
T A G C T ␣ A
```

Figura 2.1: Exemplo de dois possíveis alinhamentos entre duas sequências.

Mesmo sem uma função de pontuação definida, podemos ver pela Figura 2.1 que o segundo alinhamento expressa melhor a similaridade entre as sequências do que o primeiro alinhamento. Por convenção, quando falarmos em alinhamento, estaremos nos referindo ao alinhamento global, sendo este o foco desta dissertação.

2.1 Problema do Alinhamento de Pares de Sequências – APS

Formalmente, podemos definir o Problema do Alinhamento de Pares de Sequências como:

Definição 1 *Dadas duas sequências s e t sobre um alfabeto Σ e fixada uma matriz de pontuação c, que satisfaça os axiomas de métrica, encontrar um alinhamento A^* de s e t cuja pontuação $c(A)$ seja mínima [dB03].*

Podemos ver o problema do APS como um problema de otimização, onde queremos encontrar um alinhamento entre as duas sequências. Também podemos observar que o problema APS é um problema de otimização limitado pois, uma vez que p é métrica, todo alinhamento tem custo não-negativo, ou seja, sempre há um alinhamento que é solução do problema.

Podemos solucionar o problema APS utilizando programação dinâmica. A programação dinâmica é uma ferramenta importante para solucionar problemas de otimização tratáveis. Essa técnica soluciona um problema através da combinação das soluções dos seus subproblemas. Em geral, quando um problema de otimização possui subestrutura ótima e sobreposição de subproblemas, a técnica de programação dinâmica pode ser utilizada para solucioná-lo.

Dizemos que um problema possui *subestrutura ótima* se uma solução ótima para o problema contém soluções ótimas para os subproblemas. Além disso, quando um procedimento recursivo revisita o mesmo problema diversas vezes, dizemos que o problema de otimização tem *sobreposição de subproblemas* [CLRS09].

Para o problema APS, o algoritmo que utiliza o paradigma de programação dinâmica opera em duas etapas. Na primeira, representada pelo Algoritmo 1, para as sequências s e t de comprimentos $|s| = m$ e $|t| = n$, uma matriz A de dimensões $(m+1) \times (n+1)$, indexada por $\{0, \ldots, m\}$ e $\{0, \ldots, n\}$, é preenchida com as pontuações de alinhamentos ótimos de prefixos de s e com prefixos de t, de forma que a posição (i, j) de A contenha a pontuação de um alinhamento ótimo de $s[1, \ldots, i]$ e $t[1, \ldots, j]$, isto é, de modo que $A[i, j] = d(s[1, \ldots, i], t[1, \ldots, j])$, para $0 \leq i \leq m$ e $0 \leq j \leq n$. A recorrência para a primeira etapa pode ser vista na Equação 2.1:

$$A[i,j] = \max \begin{cases} A[i-1,j] + c(s[i],\sqcup), \\ A[i-1,j-1] + c(s[i],t[j]), \\ A[i,j-1] + c(\sqcup,t[j]) \end{cases} \tag{2.1}$$

onde $c(s[i],\ \sqcup),\ c(\sqcup,\ t[j])$ e $c(s[i],t[j])$ são os custos de se alinhar um caractere de s com espaço, um espaço com um caractere de t e um caractere de s com um caractere de t, respectivamente.

Algoritmo 1 Dist(s,t,c)

Entrada: Duas sequências s e t, com $|s| = m$ e $|t| = n$ e a matriz de pontuação c.

Saída: Uma matriz $A = (a_{ij})$ com $A[i,j] = d(s[1..i], t[1..j])$.

```
1: m ← |s|; n ← |t|; A[0,0] ← 0
2: para j = 1,...,n faça
3:    A[0,j] ← A[0,j−1] + c(⊔,t[j])
4: fim para
5: para i = 1,...,m faça
6:    A[i,0] ← A[i−1,0] + c(s[i],⊔)
7:    para j = 1,...,n faça
8:       A[i,j] ← A[i−1,j] + c(s[i],⊔)
9:       se A[i,j] > A[i−1,j−1] + c(s[i],t[j]) então
10:         A[i,j] ← A[i−1,j−1] + c(s[i],t[j])
11:      fim se
12:      se A[i,j] > A[i,j−1] + c(⊔,t[j]) então
13:         A[i,j] ← A[i,j−1] + c(⊔,t[j])
14:      fim se
15:   fim para
16: fim para
17: retorne A
```

Na segunda etapa, representada pelo Algoritmo 2, construímos os alinhamentos ótimos utilizando a tabela A resultante da primeira etapa. A construção de um alinhamento ótimo é feita observando-se qual das pontuações dentre $A[m-1,n]$, $A[m-1,n-1]$ e $A[m,n-1]$ produziu a pontuação $A[m,n]$, correspondente à pontuação ótima de todos os m caracteres de s alinhados a todos os n caracteres de t, e decidindo qual é a última coluna de um alinhamento ótimo.

Algoritmo 2 Alinha(a, s, t, c)

Entrada: Duas sequências s e t, a matriz a devolvida por $DIST(s, t)$ e a matriz de pontuação c.

Saída: Um alinhamento entre s e t de pontuação mínima.

1: $m \leftarrow |s|; n \leftarrow |t|;$
2: **se** $m = 0$ **então**
3: **retorne** os caracteres de t alinhados a espaços em s
4: **fim se**
5: **se** $n = 0$ **então**
6: **retorne** os caracteres de s alinhados a espaços em t
7: **fim se**
8: **se** $A[m, n] = A[m - 1, n] + c(s[m], \sqcup)$ **então**
9: **retorne** $Alinha(a, s[1..m - 1], t) : \begin{matrix} s[m] \\ \sqcup \end{matrix}$
10: **fim se**
11: **se** $A[m, n] = A[m - 1, n - 1] + c(s[m], t[n])$ **então**
12: **retorne** $Alinha(a, s[1..m - 1], t[1..n - 1]) : \begin{matrix} s[m] \\ t[n] \end{matrix}$
13: **fim se**
14: **se** $A[m, n] = A[m, n - 1] + c(\sqcup, t[n])$ **então**
15: **retorne** $Alinha(a, s, t[1..n - 1]) : \begin{matrix} \sqcup \\ t[n] \end{matrix}$
16: **fim se**

O Algoritmo 1 tem complexidade $O(mn)$, pois em sua execução o laço interno com início na linha 5 é executado n vezes para cada caractere da outra sequência. Já o Algoritmo 2, tem complexidade $O(m+n)$, visto que cada coluna do alinhamento construído como solução requer que, no máximo, 3 posições da matriz A sejam analisadas. Como estes testes são feitos em tempo constante e os alinhamentos considerados possuem comprimento máximo de $m+n$ colunas, temos a complexidade $O(m+n)$.

2.2 Variações do Alinhamento de Pares de Sequências

Na prática, temos três versões diferentes para o alinhamento de sequências, que são utilizados dependendo se vamos alinhar as sequências por inteiro ou se vamos alinhar apenas subsequências delas. Para tanto, temos os alinhamentos: Global, Local e Semi-global.

Alinhamento Global

O alinhamento global é o mesmo proposto para o problema do alinhamento de pares de sequências.

Alinhamento Local

Um alinhamento local entre duas sequências s e t é um alinhamento entre uma subsequência de s e uma subsequência de t e pode ser resolvido pelos Algoritmos 1 e 2 com algumas modificações na inicialização [SM97]. A recorrência para a primeira etapa pode ser vista na Equação 2.2:

$$A[i,j] = \max \begin{cases} A[i-1,j] + c(s[i], \sqcup), \\ A[i-1,j-1] + c(s[i], t[j]), \\ A[i,j-1] + c(\sqcup, t[j]) \\ 0 \end{cases} \tag{2.2}$$

onde $c(s[i], \sqcup)$, $c(\sqcup, t[j])$ e $c(s[i], t[j])$ são os custos de se alinhar um caractere de s com espaço, um espaço com um caractere de t e um caractere de s com um caractere de t, respectivamente. Com esta pequena modificação na recursão aceitamos alinhamentos de tamanho igual a 0. Para localizar o alinhamento, começamos do maior valor da matriz A e paramos quando

atingirmos uma entrada da matriz A com valor 0 ou quando essa entrada não possuir uma direção para seguir. Podemos ver um exemplo de alinhamento local das sequências `AAACCGT` e `ACC` na Figura 2.2.

Figura 2.2: Um exemplo de alinhamento local.

Alinhamento Semi-global

Um alinhamento semi-global é feito ignorando espaços no começo e no fim das sequências. O Algoritmo 1 pode ser utilizado para encontrar um alinhamento semi-global modificando a inicialização. Assim como para o alinhamento local, a primeira coluna e a primeira linha da matriz A devem ser inicializadas com 0. Já a execução do Algoritmo 2, começa no maior valor da matriz A e termina na origem da matriz, $A[0][0]$. Dois exemplo de alinhamentos semi-globais das sequências `CAGCACTTGGATTCTCGG` e `CAGCGTGG` podem ser vistos na Figura 2.3.

Figura 2.3: Dois exemplos de alinhamentos semi-globais entre duas sequências, sendo o segundo melhor que o primeiro.

Capítulo 3

Alinhamento de Várias Sequências

O Problema do Alinhamento de Várias Sequências é uma generalização do Problema do Alinhamento de Pares de Sequências, onde espaços são inseridos dentro de cada sequência de forma que as sequências resultantes tenham o mesmo tamanho e que não existam colunas compostas apenas por espaços, o que pode ser exemplificado na Figura 3.1.

```
A A G A A ␣ A
A T ␣ A A T G
C T G ␣ G ␣ G
```

Figura 3.1: Exemplo de alinhamento de várias Sequências.

Frequentemente, padrões biologicamente importantes não podem ser revelados comparando pares de sequências por vez. Com o alinhamento de várias sequências, esses padrões se tornam claros e é possível organizar as sequências em uma árvore, facilitando a visibilidade das sequências que têm alguma ligação evolucionária. Entretanto, para alinhar k sequências com tamanho n, $n \geq \max_{i=1}^{k}\{n_i\}$, temos que fazer $O(n^k)$ alinhamentos de pares de sequências, o que inviabiliza a solução do problema para um valor grande de k ou de n.

A atribuição de custos a alinhamentos de várias sequências pode ser de um número maior de formas do que a feita para pares de sequências. Um dos métodos mais utilizados para definir pontuações é atribuir uma pontuação a

cada coluna e tomar como pontuação do alinhamento a soma das pontuações de suas colunas.

3.1 Pontuação por Soma de Pares

Pontuar um alinhamento de várias sequências é mais complexo do que pontuar um alinhamento de pares de sequências e temos duas características que são desejáveis para essa função [SM97]:

1. A função deve ser independente da ordem de seus argumentos, isto é, o custo de uma coluna qualquer do alinhamento deve ser igual ao custo de uma permutação qualquer dessa coluna; e

2. A função deve associar a presença de vários caracteres semelhantes com pontuações elevadas e associar pontuações baixas quando forem encontrados caracteres diferentes e espaços.

A função de pontuação por *Soma de Pares*, ou SP, é bastante utilizada, satisfaz as características descritas e pode ser descrita como o somatório dos custos de todos os pares de símbolos da coluna. Formalmente, fixada uma função de pontuação de pares de caracteres c, definimos a função de pontuação $\mathrm{SP}_c : (\Sigma')^k \to \mathbb{Q}_{\geq 0}$ que mapeia uma coluna $\mathcal{C}$ com k caracteres à sua pontuação $\mathrm{SP}_c(\mathcal{C})$ por

$$\mathrm{SP}_c(\mathcal{C}) = \sum_{1 \leq i < i' \leq k} c(\mathcal{C}[i], \mathcal{C}[i']),$$

onde $\mathcal{C}[i]$ representa o i-ésimo caractere da coluna $\mathcal{C}$. Podemos definir, então, a pontuação $\mathrm{SP}(A)$ de um alinhamento A das k sequências $s_1, s_2, \ldots, s_k$ como

$$\mathrm{SP}(A) = \sum_{j=1}^{n} \mathrm{SP}(A[\cdot, j]) = \sum_{j=1}^{n} \sum_{1 \leq i < i' \leq k} c(A[i, j], A[i', j]),$$

onde $A[\cdot, j]$ denota a j-ésima coluna de A.

Se $s_1, s_2, \ldots, s_k$ são k sequências sobre um mesmo alfabeto Σ, definimos o custo $c(s_1, s_2, \ldots, s_k)$ de alinhar as sequências como

$$c(s_1, s_2, \ldots, s_k) = \min_{A \in \mathcal{A}_{s_1, s_2, \ldots, s_k}} \{c(A)\},$$

onde $\mathcal{A}_{s_1, s_2, \ldots, s_k}$ é o conjunto de todos os alinhamentos entre as k sequências [dB03].

3.2 Problema do Alinhamento de Várias Sequências – AVS

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

Definição 2 (Problema do Alinhamento de Várias Sequências) *Dados um inteiro $k \geq 2$ e k sequências $s_1, s_2, \ldots, s_k$ sobre um alfabeto Σ e uma função de pontuação $c : \Sigma' \times \Sigma' \to \mathbb{Q}_{\geq 0}$, encontrar um alinhamento A cujo custo $c(A)$ seja igual a $c(s_1, s_2, \ldots, s_k)$ [dB03].*

Um alinhamento A de $s_1, s_2, \ldots, s_k$ cuja pontuação $c(A)$ seja igual a $c(s_1, s_2, \ldots, s_k)$ é dito um alinhamento ótimo.

Um algoritmo utilizando o paradigma da programação dinâmica, nos moldes do algoritmo visto para o alinhamento de pares de sequências, pode ser utilizado no problema AVS.

Utilizando uma matriz A com k dimensões, onde sua i-ésima dimensão é indexada de 0 ao comprimento n_i da i-ésima sequência da entrada e armazena o custo do melhor caminho de $\overrightarrow{0}$ a um vértice $(i_1, \ldots, i_k)$, onde $\overrightarrow{0}$ é um vetor composto por 0's.

O Algoritmo 3 testa, a cada iteração, qual das colunas pode ser concatenada a um alinhamento ótimo para um subproblema, a fim de obter a solução para um subproblema maior. A pontuação da coluna $a(i_1, \ldots, i_k)$ é equivalente a 0, quando $\overrightarrow{i} = \overrightarrow{0}$, e para $\overrightarrow{i} \neq \overrightarrow{0}$ temos:

$$a[\overrightarrow{i}] = \min_{b \in \{0,1\}^k \setminus \{\overrightarrow{0}\} \ e \ b \leq \overrightarrow{i}} \{a[\overrightarrow{i} - b] + \mathrm{SP}(b \otimes s[\overrightarrow{i}])\},$$

onde b é um vetor binário, tal que $b \neq 0$, e $(b \otimes s[\overrightarrow{i}])$ é o custo da coluna tal que:

$$b \otimes s[\overrightarrow{i}] = \begin{cases} s[i] & se\ b = 1 \\ \sqcup & se\ b = 0 \end{cases}$$

A complexidade de tempo gasta pelo Algoritmo 3 é $O(2^k k^2 \prod_{i=1}^{k}(n_i + 1))$, visto que o algoritmo executa um laço que calcula o mínimo de $2^k - 1$ termos, no máximo, e é executado $\prod_{i=1}^{k}(n_i + 1)$ vezes. Cada termo pode ser computado em tempo $O(k^2)$, devido à computação da função SP para uma

Algoritmo 3 Dist-AVS($k, s_1, ..., s_k$)

Entrada: Um inteiro $k \geq 2$ e k sequências $s_1, \ldots, s_k$.
Saída: Uma matriz A com $A[i_1, \ldots, i_k] = \mathrm{SP}(s_1[1..i_1], \ldots, s_k[1..i_k])$.

1: $A[0, \ldots, 0] \leftarrow 0$
2: **para** $\overrightarrow{i} \in \{\overrightarrow{0}, \ldots, \overrightarrow{n}\} \backslash \{\overrightarrow{0}\}$ em ordem lexicográfica **faça**
3: $\quad A[\overrightarrow{i}] \leftarrow \min_{b \in \{0,1\}^k \backslash \{\overrightarrow{0}\} \ e \ b \leq \overrightarrow{i}} \{A[\overrightarrow{i} - b] + \mathrm{SP}(b \otimes s[\overrightarrow{i}])\}$
4: **fim para**
5: **retorne** A

coluna de k posições e para somar este valor ao valor de uma entrada de A. Com isso, podemos ver que o método da programação dinâmica com soma de pares só é prático para alinhamentos com um pequeno número de sequências.

3.3 Clustal W

Como vimos na Seção 3.2, a versão do algoritmo usando o paradigma da programação dinâmica não é eficiente para um grande número de sequências ou para sequências muito grandes. Como alternativa a esse paradigma, temos heurísticas para alinhar várias sequências, como o alinhamento progressivo, que produz o alinhamento de várias sequências a partir de vários alinhamentos par a par [dS08]. Como podemos ver na Figura 3.2, o alinhamento progressivo é composto por três estágios: cálculo da matriz de distâncias, construção da árvore guia e alinhamento progressivo, sendo que os métodos utilizados por esses estágios podem variar de acordo com o algoritmo que utiliza essa heurística.

Utilizando essa heurística temos os programas *T-Coffee* [NHH00], *MUSCLE* [Edg04] e *Clustal W* [THG94]. Desses, um dos mais populares para se resolver o Problema do Alinhamento de Várias Sequências é o algoritmo utilizado pelo programa *Clustal W* e, como um programa que utiliza a heurística de alinhamento progressivo, vai adicionando as sequências uma a uma a um alinhamento existente, criando um novo alinhamento. A ordem com a qual as sequências são adicionadas ao alinhamento é definida por uma árvore guia, que é computada utilizando o custo do alinhamento entre os possíveis alinhamentos dos pares de sequências.

Para k sequências de tamanho n, temos que o primeiro estágio do algo-

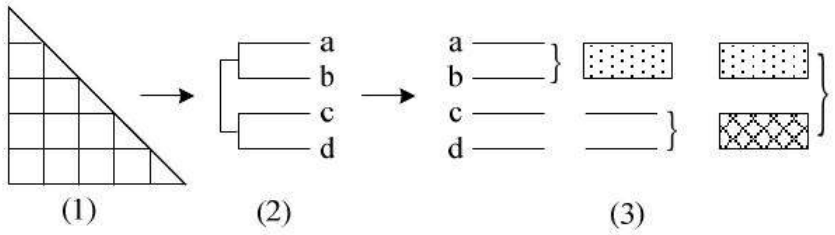


Figura 3.2: Os três estágios do alinhamento progressivo: (1) Cálculo da matriz de distâncias; (2) Construção da árvore guia; e (3) Alinhamento progressivo.

ritmo tem complexidade $O(k^2n^2)$ para calcular a matriz de distâncias. No segundo estágio, a construção da árvore guia tem complexidade $O(k^4)$, enquanto que o terceiro estágio tem complexidade $O(k^3 + kn^2)$ para construir o alinhamento das várias sequências. Com isso, o *Clustal W* tem complexidade total de $O(k^4 + n^2)$ [Edg04].

A seguir detalhamos cada um dos 3 estágios do *Clustal W*.

3.3.1 Matriz de Distâncias

No primeiro estágio, calculamos a matriz de pontuação das sequências alinhando todas as sequências duas a duas, utilizando o algoritmo de programação dinâmica, de tal forma que, para k sequências, $k(k-1)/2$ alinhamentos serão calculados.

```
(1)  A A G A A A
(2)  A T A A T G
(3)  G A C G G A T T A G
(4)  G A T C G G A A T A G
(5)  C T G G G
(6)  C A G C A C T T G G A T T C T C G G
(7)  C A G C G T G G
```

Figura 3.3: Sequências de entrada do *Clustal W*.

Como exemplo, temos na Figura 3.3 as sequências que serão alinhadas pelo algoritmo do *Clustal W*. Na Tabela 3.1, temos o resultado da aplicação do cálculo da matriz de distâncias nas sequências, usando, como matriz de

pontuação, a matriz de distância de edição 0 ou 1 e penalidade para o *gap* igual a 2.

		1	2	3	4	5	6	7
1	..	-	3	5	9	11	25	7
2	..	3	-	5	10	10	25	7
3	..	5	5	-	12	13	26	7
4	..	9	10	12	-	3	17	8
5	..	11	10	13	3	-	17	9
6	..	25	25	26	17	17	-	20
7	..	7	7	7	8	9	20	-

Tabela 3.1: Matriz de distâncias D.

3.3.2 Árvore Guia

O segundo estágio realiza a construção da árvore guia ou filogenética com a informação fornecida pela matriz de pontuação. Neste estágio do algoritmo, determinamos a ordem dos alinhamentos, que serão passados para o próximo estágio, com a ideia de tentar encontrar uma árvore que relacione as sequências de uma forma evolucionária.

Existem vários métodos que podem ser utilizados na construção de árvores filogenéticas, estes estão divididos em dois grupos os métodos baseados em distâncias, *Neighbor-Joining*(NJ) [SN87], *QuickTree* [HBD02] e *RapidNJ* [SMP08], e os métodos baseados em evolução mínima (*Maximum-Likelihood*), *PhyML* [GG03] e *FastTree* [PDA09].

Os métodos baseados em distâncias tem uma complexidade menor e o método NJ é utilizado pelo algoritmo *Clustal W*. Por esse motivo, este texto abordará os métodos baseados em distâncias, mais especificamente nas implementações do NJ, *RapidNJ* e *NINJA* [Whe09], que serviram de base para o algoritmo utilizando MPI.

Nesse contexto, o objetivo do método *Neighbor-Joining* (NJ) é construir uma árvore topológica e obter o tamanho dos ramos na árvore final juntando as sequências, que representam, cada uma, unidades taxonômicas operacionais (*OTU's*), vizinhas.

Neighbor-Joining (NJ)

Um par de *OTU's* são vizinhos quando compartilham um único nó interior em uma árvore bifurcada e sem raiz. O número de pares de vizinhos é proporcional à topologia da árvore e, para uma árvore com $N \geq 4$ *OTU's*, o número mínimo de vizinhos é sempre 2 e o número máximo pode ser $N/2$, se N for par, ou $(N-1)/2$, se N for ímpar.

Este método começa com uma árvore estrela e procede fazendo junções sucessivas nas *OTU's* vizinhas até encontrar $N-3$ *OTU's*.

O primeiro passo do método consiste em identificar o par de *OTU's* tal que a árvore tenha a menor soma de ramos. O que pode ser determinado aplicando a Equação 3.1 em todos os pares de *OTU's*:

$$S_{ij} = (N-2)D_{ij} - (R_i + R_j) \tag{3.1}$$

onde $R_i = \sum_k D_{ik}$, o que pode ser exemplificado aplicando a Equação 3.1 na Tabela 3.1, resultando na Tabela 3.2. Com os valores de i e j tais que S_{ij} seja o menor, podemos adicionar uma nova OTU u, que é composta por i e j, com novos valores para a matriz de distâncias. A Equação 3.2 é utilizada para calcular os novos valores da matriz de distâncias entre a nova OTU u e as demais que não fazem parte de u e as equações 3.3 e 3.4 são utilizadas para o cálculo da distância entre i e u e entre j e u, como são descritas em [SK88].

$$D^l_{uk} = \frac{1}{2}(D^{l-1}_{ik} + D^{l-1}_{jk} - D^{l-1}_{ij}), \qquad para\ k \neq i, j. \tag{3.2}$$

$$D^l_{iu} = \frac{1}{2(N-2)}[(N-2)D^{l-1}_{ij} + R_i - R_j]. \tag{3.3}$$

$$D^l_{ju} = \frac{1}{2(N-2)}[(N-2)D^{l-1}_{ij} + R_j - R_i]. \tag{3.4}$$

Na Tabela 3.3, temos a primeira iteração do método, onde as *OTU's* 5 e 6 foram agrupadas. O resultado dessa junção pode ser visto na Figura 3.4b.

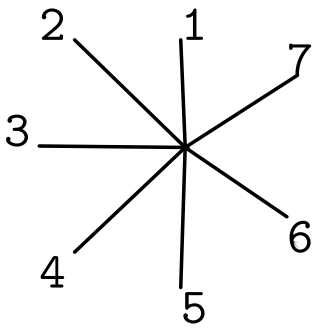
Esse método gera uma árvore sem raiz com ramos proporcionais à divergência estimada entre cada ramo, o que pode ser visualizado na Figura 3.4e. A raiz da árvore é colocada na posição onde os dois lados da árvore possuem o mesmo tamanho, resultando na Figura 3.4f.

	OTU						
OTU	1	2	3	4	5	6	7
1 ..	-	-105.0	-103.0	-74.0	-68.0	-65.0	-83.0
2 ..	-105.0	-	-103.0	-69.0	-73.0	-65.0	-83.0
3 ..	-103.0	-103.0	-	-67.0	-66.0	-68.0	-91.0
4 ..	-74.0	-69.0	-67.0	-	-107.0	-104.0	-77.0
5 ..	-68.0	-73.0	-66.0	-107.0	-	-108.0	-76.0
6 ..	-65.0	-65.0	-68.0	-104.0	-108.0	-	-88.0
7 ..	-83.0	-83.0	-91.0	-77.0	-76.0	-88.0	-

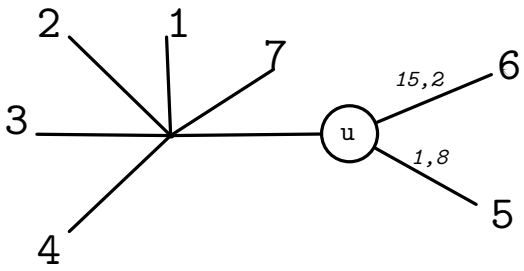
Tabela 3.2: Matriz S com o tamanho dos ramos da árvore de NJ no primeiro ciclo.

	OTU					
OTU	1	2	3	4	5-6	7
1 ..	-	-55.5	-53.5	-38.0	-32.5	-40.5
2 ..	-55.5	-	-54.0	-34.5	-35.0	-41.0
3 ..	-53.5	-54.0	-	-32.5	-33.0	-47.0
4 ..	-38.0	-34.5	-32.5	-	-71.5	-43.5
5-6 ..	-32.5	-35.0	-33.0	-71.5	-	-48.0
7 ..	-40.5	-41.0	-47.0	-43.5	-48.0	-

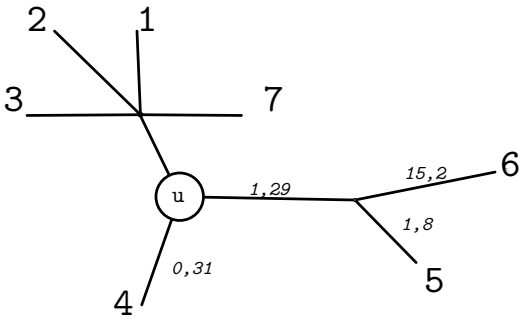
Tabela 3.3: Matriz S' com o tamanho dos ramos após a junção das *OTU's* 5 e 6.



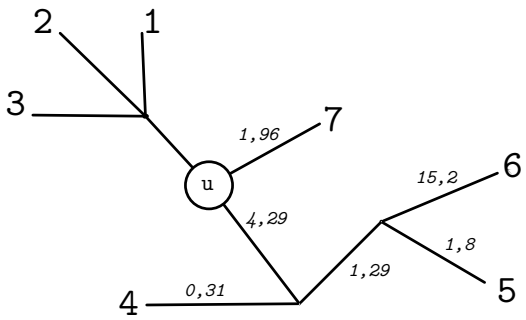
(a) Estado inicial da árvore.



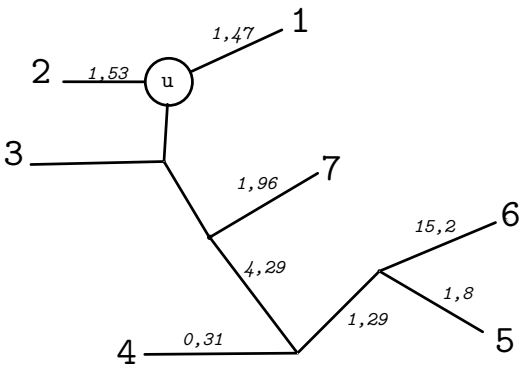
(b) Junção das *OTU's* 5 e 6.



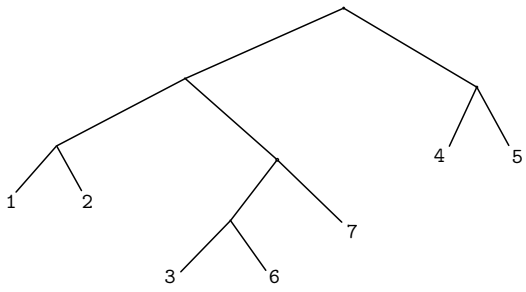
(c) Árvore com junção das *OTU's* 5-6 e 4.



(d) Árvore com junção das *OTU's* 4-5-6 e 7.



(e) Junção das *OTU's* 1 e 2.



(f) Árvore guia final.

Figura 3.4: Passo a passo da construção da árvore filogenética com o tamanho dos ramos em itálico.

RapidNJ

Esta implementação, como visto em [SMP08], mantêm o pior caso com complexidade $O(n^3)$, mas oferece uma melhora substancial na prática em relação ao NJ. Isto é feito acelerando a procura pelos pares de nó que serão agrupados, mantendo os critérios utilizados pelo método original, e se baseia na seguinte observação:

> Quando se está procurando o menor valor de S, s_{min}, na Equação 3.1, R_i é constante no contexto da linha i [SMP08].

Com essa observação é possível construir um limite superior nas procuras por s_{min} que é dinamicamente atualizado de acordo com os valores de S. Também é necessário criar duas novas estruturas para utilizar esse limite superior: Q, uma matriz contendo triplas de $(\{i, j\}, D(i, j))$ ordenada crescentemente pelo valor de $D_{i,j}$, e I, uma matriz que mapeia as posições de Q para D. Seja $o_1, o_2, \ldots, o_n$ uma permutação de $1, 2, \ldots, n$ tal que $D_{i,o_1} \leq D_{i,o_2} \leq \ldots \leq D_{i,o_n}$, então:

$$Q(i, j) = D_{i,o_j}, \tag{3.5}$$

e

$$I(i, o_j) = j. \tag{3.6}$$

O RapidNJ começa calculando todos os valores para R de tal forma que ele ache R_{max} e, então, o algoritmo segue da seguinte forma:

1. $s_{min} = \infty$, $i = -1$, $j = -1$
2. para cada linha l de S e para cada coluna c de l:
   (a) se $S(l,\ c) - R(l) - R_{max} > s_{min}$ então avance para a próxima linha.
   (b) se $Q(l,\ I(l,\ c)) < s_{min}$ então $s_{min} = Q(l,\ I(l,\ c))$, $i = l$ e $j = I(l,\ c)$.

Tendo escolhido os dois nós que serão agrupados o algoritmo atualiza a matriz D da mesma forma que o método NJ. A remoção dos dois nós não é feita em Q, os nós são marcados como removidos e permanecem lá até que o espaço seja necessário. O novo nó criado pelo agrupamento é inserido em Q e em I e Q é novamente ordenada.

NINJA

A abordagem utilizada pelo *RapidNJ* é correta e produz uma melhora substancial na prática [SMP08], entretanto essa abordagem ainda pode ser melhorada. O limite superior do *RapidNJ* é dependente de R_{max}, o que pode ser muito frouxo [Whe09]. Para deixar esse limite mais justo, podemos dividir o intervalo $(R_{min},\ R_{max})$ em compartimentos disjuntos, onde cada compartimento B_x cobre o intervalo $[R_x^{min},\ R_x^{max})$. Para X compartimentos, o intervalo entre mínimo e máximo será $(R_{max} - R_{min})/X$. Cada *OTU's* i é associada com um compartimento B_x para o qual $R_x^{min} \leq R_i < T_x^{max}$. Podemos criar, então, um conjunto $S_{\{x,y\}}$ para cada par de compartimentos $\{B_x, B_y\}$. Com isso, o cálculo do limite para o conjunto $S_{\{x,y\}}$ pode ser descrito por:

$$s_{bound} = (N-2)D_{ij} - R_x^{max} - R_y^{max} \tag{3.7}$$

A Equação 3.7 melhora o filtro porque, para um par não visitado $\{i', j'\}$ do mesmo conjunto $S_{\{x,y\}}$, $(N-2)D_{i'j'} - R_x^{max} - R_y^{max}$ vai ser um limite mais justo do que $(N-2)D_{i'j'} - R_{i'} - R_{max}$.

Após agrupar os nós i e j, as linhas e colunas associadas com esses nós são marcadas como inativas em D e uma nova linha e coluna é adicionada para o nó agrupado ij. Este novo nó fica associado com o compartimento $B(ij) = argmin_x\{R_x^{max} > R_{ij}\}$. As triplas $(\{ij, k\}, D_{ij|k})$ para os nós remanescentes são adicionados nos conjuntos apropriados, $S_{B(ij),B(k)}$ e as triplas dos nós i e j são removidas.

Como as estruturas de dados utilizadas são *heaps*, na procura por s_{min}, triplas $(\{i, j\}, D_{ij})$ são removidas de seus *heaps*. Podemos chamar esses triplas de *candidatas*. O método reduz drasticamente o número de candidatos vistos na maioria dos casos, mas é extremamente ineficiente quando as sequências tem um parentesco maior. Por esse motivo, um outro filtro foi implementado sobre os candidatos e funciona pegando candidatos que passaram pelo filtro anterior e os organizando de uma forma que o novo limite diminua o número de candidatos que são vistos em cada iteração.

Sejam $s_{ij}(p)$, $N(p)$ e $R_i(p)$ correspondente aos valores de s_{ij}, N e R_i em uma iteração previamente fixada p e seja $\delta_i(p) = (N-2)R_i(p) - (N(p)-2)R_i$. Então é fácil mostrar para essa iteração que:

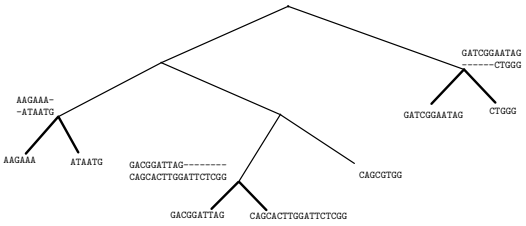
$$s_{ij} = \frac{(N-2)s_{ij}(p) + \delta_i(p) + \delta_j(p)}{N(p) - 2} \tag{3.8}$$

Suponha que todos os candidatos na iteração p são armazenados como triplas $(\{i, j\}, s_{ij}(p))$ em um conjunto de candidatos ordenado pelo valor de $s_{ij}(p)$. Dados os valores de $N(p)$ e $R_i(p)$ par o conjunto, o par $\{i, j\}$ com o menor s_{ij} para uma próxima iteração vai estar próximo ao primeiro elemento do conjunto.

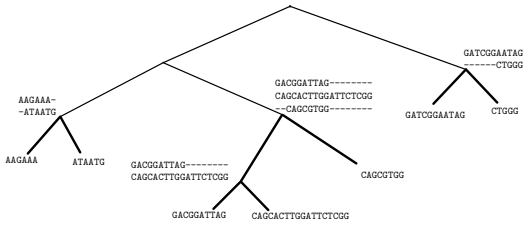
Esse método aumenta o ganho de performance obtido pelo método *RapidNJ*, diminuindo drasticamente o número de candidatos vistos, porém, a complexidade continua sendo $O(n^3)$ no pior caso.

3.3.3 Alinhamento Progressivo

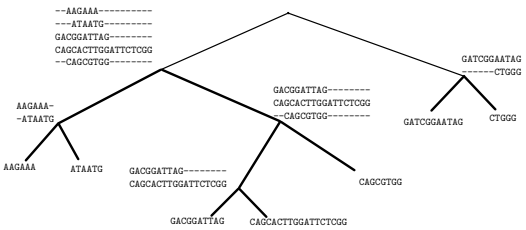
No último estágio do algoritmo, realizamos o alinhamento progressivo das sequências. Com as sequências ordenadas pelo segundo estágio, esta parte do algoritmo alinha grupos cada vez maiores de sequências da árvore guia partindo dos nós folhas, como pode ser visto na Figura 3.5a, e indo em direção à raiz, seguindo a ordem em que os grupos aparecem na árvore guia, como podemos ver na sequência de Figuras 3.5b, 3.5c e 3.5d.



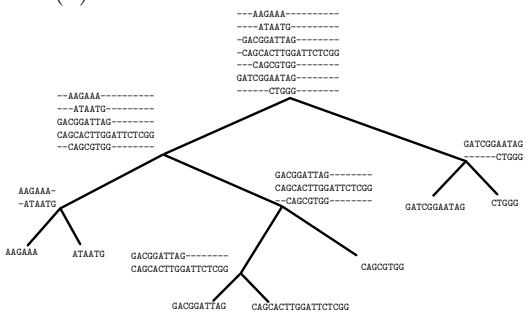
(a) Início pelos nós folhas.



(b) Alinhamento das OTU's 3-6 e 7.



(c) Alinhamento das OTU's 1-2 e 3-6-7.



(d) Último alinhamento feito na raiz.

Figura 3.5: Passo a passo do alinhamento prgressivo, onde os alinhamentos, em negrito, são feitos partindo dos nós folhas.

E, para a matriz de distâncias da Tabela 3.1, o alinhamento progressivo pode ser visto na Figura 3.6.

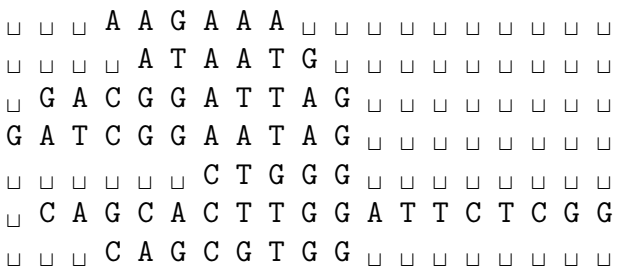


Figura 3.6: Resultado do Alinhamento das Várias Sequências.

Cada alinhamento desse estágio é feito utilizando programação dinâmica, mantendo espaços inseridos e adicionando novos espaços nas sequências quando necessário. Para alinhar as k sequências utilizando o resultado do estágio 2 são necessários $k - 1$ alinhamentos.

Capítulo 4

Modelos e Plataformas de Computação Paralela

Com o aumento no número de sequências que serão alinhadas ou no tamanho dessas sequências, o tempo gasto para alinhar as várias sequências vai se tornando cada vez maior. Para diminuir o tempo gasto nesses alinhamentos com um número muito grande de sequências, paralelizações de soluções já existentes foram feitas. Uma solução que já foi paralelizada, com granulosidade grossa e com granulosidade fina, é a proposta pelo *Clustal W*.

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

Os modelos realísticos de computação paralela surgem através da busca de um modelo adequado com características intrínsecas à computação paralela. Esses modelos podem ser atingidos em um ambiente de inovações tecnológicas aceleradas, apesar de muito abstratos, e deveriam balancear simplicidade com precisão e abstração com praticidade [Ste03].

4.1 Modelo PRAM

O modelo PRAM (*Parallel Random Access Machine*) [FW78] foi um dos primeiros modelos de computação paralela a ser idealizado e consiste de vários processadores sequenciais independentes, cada um com sua memória privada, fazendo comunicação através de uma memória global, o que pode ser visto

na Figura 4.1. Este modelo pode ser classificado de acordo com as restrições de acesso à memória global, podendo ser:

EREW PRAM a leitura e a escrita simultânea à uma posição da memória por dois processadores não é permitida;

CREW PRAM leituras simultâneas são permitidas e escritas simultâneas não são permitidas; e

CRCW PRAM leitura e escrita simultâneas são permitidas, sendo que, nesse caso, os conflitos de escrita devem ser resolvidos por um dos métodos:

1. Todos os processadores escrevem na mesma localização da memória o mesmo valor;
2. Um dos processadores participantes da escrita simultânea tem sucesso e o algoritmo deve continuar independente do processador que obteve sucesso; e
3. Existe uma ordem linear de escrita dos processadores e o processador com menor *id* escreve seu valor.

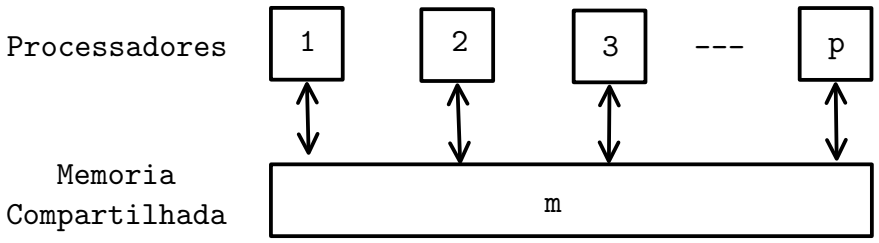


Figura 4.1: Modelo PRAM.

4.2 Modelo BSP

O modelo BSP (*Bulk Synchronous Parallel*) [Val90] é um dos principais modelos realísticos. Este modelo de granulosidade grossa é o pioneiro em incorporar os custos de comunicação, através de parâmetros, como características do modelo. O objetivo principal deste modelo é servir de *modelo*

ponte entre o desenvolvimento de algoritmos e o mundo do *hardware*, sendo este um requisito fundamental para um modelo que deseja desempenhar o mesmo papel do modelo RAM na computação sequencial.

Uma máquina BSP consiste de p processadores, cada um com sua memória local. Os processadores se comunicam através de algum meio de interconexão, gerenciados por um *roteador* com facilidade de sincronização periódica global. Um algoritmo BSP consiste numa sequência de super passos separados por barreiras de sincronização. Em um super-passo, cada processador recebe um conjunto de operações independentes, consistindo de uma combinação de passos de computação, usando dados disponibilizados localmente no início do super-passo, e passos de comunicação, através de instruções de envio e recebimento de mensagens. A resposta a uma mensagem enviada em um super-passo somente será utilizada no próximo super-passo, o que pode ser visto na Figura 4.2.

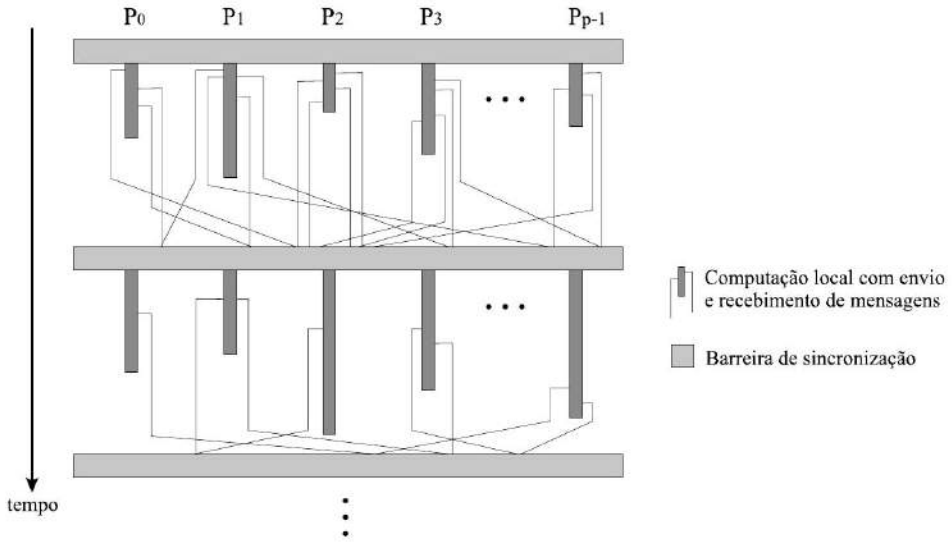


Figura 4.2: Modelo BSP.[Lou10]

O tempo de comunicação em um algoritmo no modelo BSP é dado por uma função do custo com dois parâmetros:

L descreve a periodicidade, ou seja, o tempo mínimo entre duas barreiras de sincronização; e

g descreve a taxa de eficiência entre computação e comunicação, isto é, a

razão entre o número de operações computacionais e o número total de mensagens de tamanho fixo entregues por segundo pelo roteador.

Assim, o tempo de execução de cada super-passo s é dado por $s = w + gh + L$, onde w é a maior computação realizada e h é o maior número de mensagens enviadas ou recebidas por algum processador durante o super-passo.

4.3 Modelo CGM

O modelo CGM (*Coarse Granularity Multicomputer*) [DFRC93] é uma simplificação do modelo BSP. No CGM, os algoritmos realizam sequências de super passos, onde cada super-passo é dividido em uma fase de comunicação global e uma fase de computação local. A Figura 4.3 exemplifica o modelo.

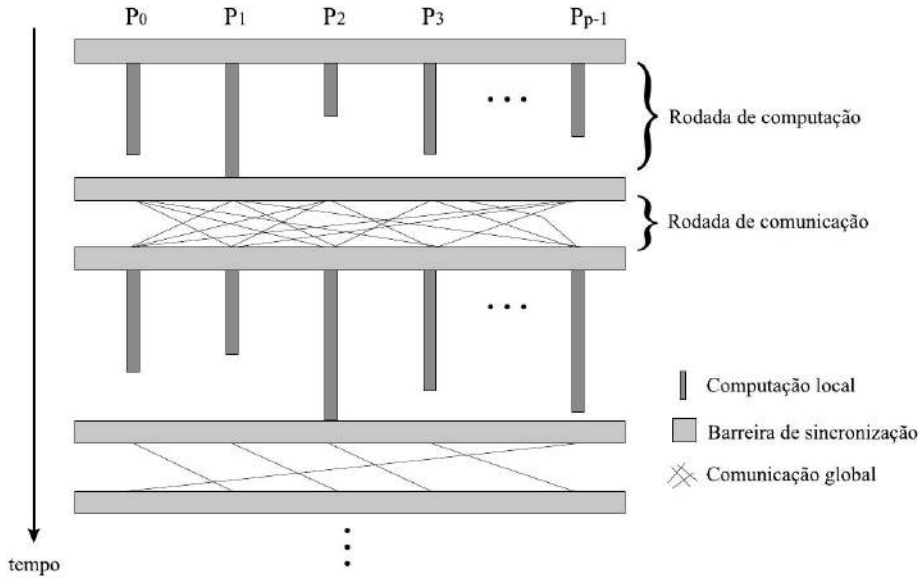


Figura 4.3: Modelo CGM.[Lou10]

Um CGM(n, p) consiste de p processadores, sendo o tamanho total da memória $O(n)$, onde n é o tamanho do problema e cada processador possui $O(n/p)$ memória local. Os processadores podem ser conectados através de um meio de intercomunicação qualquer, rede de interconexão ou memória compartilhada.

4.4 MPI

MPI (*Message-Parssing Interface*) [mpia] é uma especificação de funções e macros, que podem ser utilizadas em programas em C, C++, Fortran, Python e Java, para bibliotecas de troca de mensagens, onde as mais conhecidas são o MPICH [mpib] e o OpenMPI [GFB+04]. O objetivo do MPI é ser usado em programas que exploram a existência de múltiplos processadores através da troca de mensagens e tem como vantagens a portabilidade e a fácil utilização.

Na especificação do MPI, os processadores são identificados por uma sequência de inteiros não-negativos, ou seja, se há p processadores, então cada um será identificado por um valor entre $0, 1, \ldots, p$. Além disso, cada processador pode executar um programa diferente, permitindo que múltiplos programas operem sobre múltiplos dados, oferecendo flexibilidade tanto para programas com memória compartilhada ou distribuída.

4.5 OpenGL

A OpenGL pode ser definida como uma interface em software para fazer gráficos em hardware [SSKLK13], ou seja, é uma biblioteca de modelagem 3D que foi desenvolvida como uma interface simplificada e independente do hardware, de tal maneira que ela pode ser implementada em vários tipos diferentes de hardware ou em software no caso do hardware ser inexistente. A OpenGL foi implementada como um sistema cliente-servidor, onde a aplicação é o cliente e o servidor é a implementação do OpenGL no hardware utilizado.

A OpenGL implementa um pipeline de renderização, o que é uma sequência de passos para converter os dados da aplicação em uma imagem final. Como pode ser visto na Figura 4.4, a OpenGL começa com os dados geométricos fornecidos e os processa através de uma sequência de passos de *shading*, que é um método para desenhar objetos de forma que eles tenham

profundidade, antes de passar pelo renderizador. Este irá gerar fragmentos para cada primitiva e executar um *shader* de fragmentação para cada fragmento.

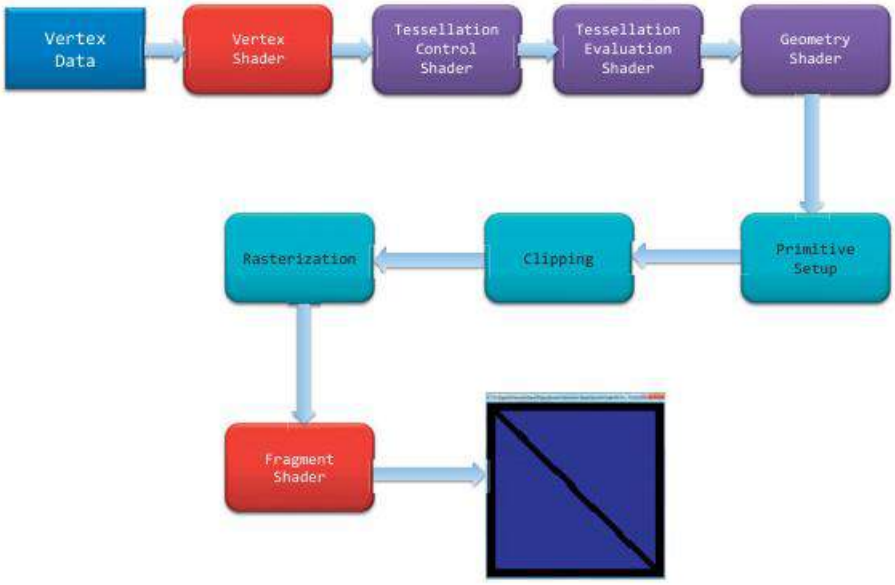


Figura 4.4: Pipeline de renderização do OpenGL.[SSKLK13]

Apesar de ter sido desenvolvida apenas para renderizar gráficos, a OpenGL também foi utilizada para implementar soluções paralelas utilizando seu pipeline de renderização. Essa forma não gráfica de utilizar as arquiteturas otimizadas para gráficos das GPUs gerou as GPUs de propósito geral (GPGPUs).

4.6 OpenCL

OpenCL é uma biblioteca que permite a implementação de programas de computadores compostos por CPUs, GPUs, FPGAs (Field-Programmable Gate Array), que são dispositivos reprogramáveis, não sendo restritos a funções pré-compiladas em hardware, e outros processadores, ou seja, é uma biblioteca para implementação de programas de computação heterogênea [MGM+12]. Com isso, podemos utilizar um único programa em uma

variedade de sistemas, celulares, notebooks e nós em *clusters* de computadores. Entretanto, a implementação do programa deve, explicitamente, definir a plataforma, o contexto e o trabalho que será realizado em cada dispositivo. Podemos ver os como são utilizados os dispositivos por um computador na Figura 4.5.

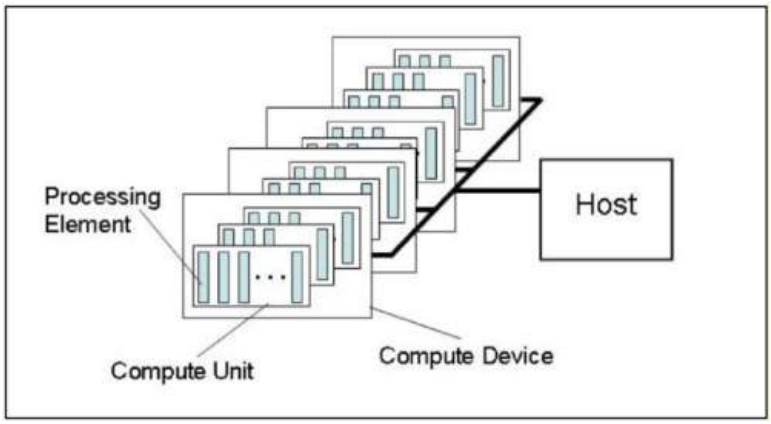


Figura 4.5: Arquitetura OpenCL.[MGM+12]

4.7 CUDA

CUDA (*Compute Unified Device Architecure*) é uma extensão do C/C++, desenvolvida pela NVIDIA, que permite que sejam feitos programas escaláveis para GPUs habilitadas [LSM09], onde os elementos computacionais dos algoritmos são chamados *kernels* e cada aplicação pode ter um ou vários *kernels*, sendo que estes devem ser escritos em uma linguagem estendida com palavras-chave adicionais para expressar paralelismo.

Um programa CUDA é composto por uma parte sequencial que é executada na CPU e uma parte paralela que é executada nos *kernels* da GPU. Por sua vez, um *kernel* é composto por um conjunto de *threads* que estão organizadas em *grids*. Cada *grid* é composto por blocos de *threads* que são sincronizadas e podem fazer comunicação entre si através de uma memória compartilhada por bloco. *Threads* de diferentes blocos podem se comunicar através de acessos atômicos à memória global, que é compartilhada por todas as *threads* [LSM11], o que pode ser visto na Figura 4.6.

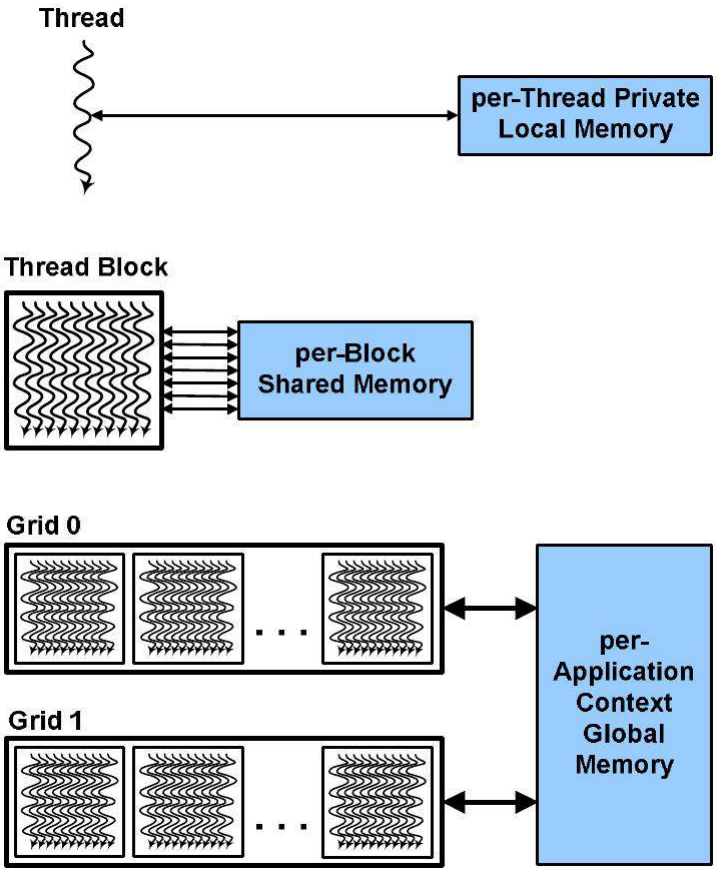


Figura 4.6: Hierarquia das *threads*, blocos e *grids*, com o seu respectivo nível de acesso à memória.[Gla09]

Uma vez compilado, os *kernels* consistem de muitas *threads* que executam o mesmo código em paralelo, onde cada *thread* pode ser considerada como uma interação em um laço. As *threads* de um bloco podem executar blocos diferentes de um mesmo programa. Porém, a maior eficiência e o melhor desempenho ocorrem quando as *threads* seguem o mesmo fluxo de execução de um programa [Gla09].

Capítulo 5

Alinhamentos de Várias Sequências em Paralelo

Existem várias paralelizações de algoritmos para o alinhamento de várias sequências, algumas de granulosidade grossa, tais como o *ClustalW-MPI* e o *ClustalW-SMP*, e outras para granulosidade fina, tais como o *MT-ClustalW* [CKT06] e o *GPU-ClustalW* [LSVMW06]. Além dessas versões, existem a MASON [DE06] e a MSA-CUDA [LSM09], de granulosidades grossa e fina, respectivamente, que serão utilizadas como base para a solução de granulosidade híbrida desenvolvida em nosso trabalho. Outras soluções para o problema do alinhamento de várias sequências também existem. O *T-Coffee* [NHH00] e o *MUSCLE* [Edg04] foram paralelizados com soluções de granulosidade grossa.

5.1 MASON

O MASON (*Multiple Alignment of Sequences Over a Network*) é uma implementação do Clustal W utilizando o MPI. Essa implementação paraleliza o primeiro e o terceiro Estágios do algoritmo, sendo que o segundo Estágio não é paralelizado por necessitar de paralelização com granulosidade fina. A seguir, descrevemos como são realizados cada um dos 3 Estágios do MASON.

5.1.1 Matriz de Distâncias

No capítulo 3.3.1 vimos que para k sequências existem $k(k-1)/2$ alinhamentos. Logo, para p processadores, serão realizados $k(k-1)/2p$ alinhamentos por processador, onde o método utilizado na implementação do MASON foi enviar todas as k sequências para todos os processadores e cada processador calcula $k(k-1)/2p$ alinhamentos.

5.1.2 Árvore Guia

A construção da árvore guia no MASON é realizada sequencialmente, pois a paralelização desse Estágio do algoritmo necessita de uma paralelização com granulosidade fina. A paralelização com granulosidade grossa tem um alto custo durante a troca de mensagens, o que inviabiliza este tipo de paralelização.

5.1.3 Alinhamento Progressivo

Como mencionado na seção 3.3.3, para alinhar k sequências são necessários $k-1$ alinhamentos e, geralmente, é possível fazer os alinhamentos dos nós da árvore guia que estão no mesmo nível em paralelo, se os alinhamentos que serão utilizados já foram realizados. Nesse Estágio, o número de computadores que serão utilizados para alinhar um nível da árvore é igual ao número de nós no nível. Dessa forma, apenas o alinhamento da raiz da árvore é feito sequencialmente.

5.2 MSA-CUDA

Assim como o MASON, o MSA-CUDA é uma implementação paralela do Clustal W, sendo que a diferença reside no fato de o MSA-CUDA ter sido desenvolvido utilizando CUDA, ou seja, é uma solução de granulosidade fina, voltada para GPUs e essa implementação consegue paralelizar todos os Estágios do Clustal W. A seguir descrevemos como são realizados cada um dos 3 Estágios do MSA-CUDA.

5.2.1 Matriz de Distâncias

Dadas duas sequências S_a e S_b de tamanhos l_a e l_b, respectivamente. A distância $d(S_a, S_b)$ pode ser definida como:

$$d(S_a, S_b) = -\frac{nid(S_a, S_b)}{\min\{l_a, l_b\}} \tag{5.1}$$

onde $nid(S_a, S_b)$ denota o número de casamentos perfeitos no alinhamento local ótimo de S_a e S_b e pode ser computado em espaço linear usando três passos, como visto em [LSM09]:

Primeiro passo: execução do algoritmo para o alinhamento local utilizando programação dinâmica;

Segundo passo: execução reversa do algoritmo para o alinhamento local utilizando programação dinâmica; e

Terceiro passo: computação do alinhamento em espaço linear utilizando o algoritmo de Myers-Miller [MM88].

Essa forma de cálculo se deve as dependências nas células da matriz, onde cada célula depende das células vizinha acima, à esquerda e na diagonal esquerda para cima, o que pode ser visto na Figura 5.1. Porém, a execução pode ser feita pelas anti-diagonais, como visto na Figura 5.2, começando do topo mais à esquerda da matriz indo em direção ao lado inferior mais à direita da matriz.

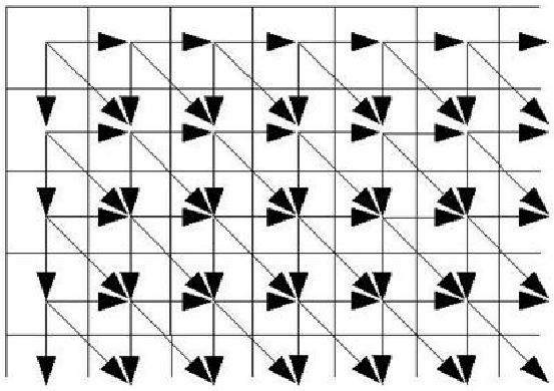


Figura 5.1: Dependências para o cálculo de uma célula na matriz.

O algoritmo de Myers-Miller encontra o alinhamento ótimo encontrando o ponto médio ótimo de um alinhamento usando as passagens do algoritmo de programação dinâmica para o alinhamento local e calculando, recursivamente, pontos médios de cada lado desse ponto ótimo, o que pode ser visto na Figura 5.3.

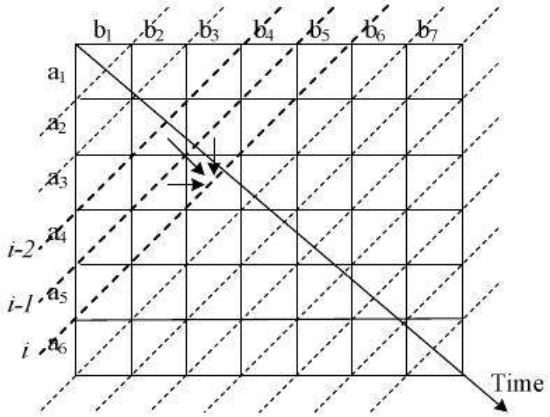


Figura 5.2: Células na mesma anti-diagonal (linhas tracejadas) podem ser computadas paralelamente.

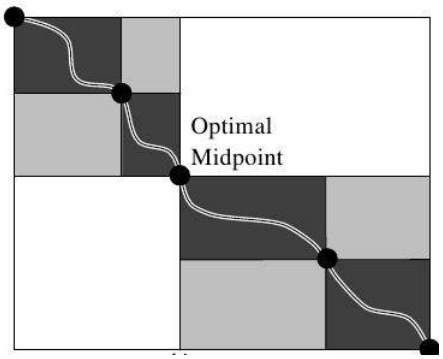


Figura 5.3: Formulação do problema em subproblemas para encontrar o alinhamento.

5.2.2 Árvore Guia

Na segunda fase do Clustal W, a construção da árvore guia é construída utilizando o método Neighbor-Joining (NJ), como é descrito na seção 3.3.2, e pode ser dividido em dois sub-estágios:

Sub-estágio 2a: Reconstrução da árvore de NJ sem raiz; e

Sub-estágio 2b: Enraização da árvore de NJ e cálculo dos pesos das sequências.

Após a reconstrução da árvore de NJ, o sub-estágio 2b começa a re-enraizar a árvore de NJ sem raiz, a recalcular o peso das sequências e a atravessar a árvore com raiz para identificar os alinhamentos do próximo Estágio. A raiz é colocada quando os pesos dos ramos são idênticos em ambos os lados do nó, o que pode ser visto na Figura 5.4.

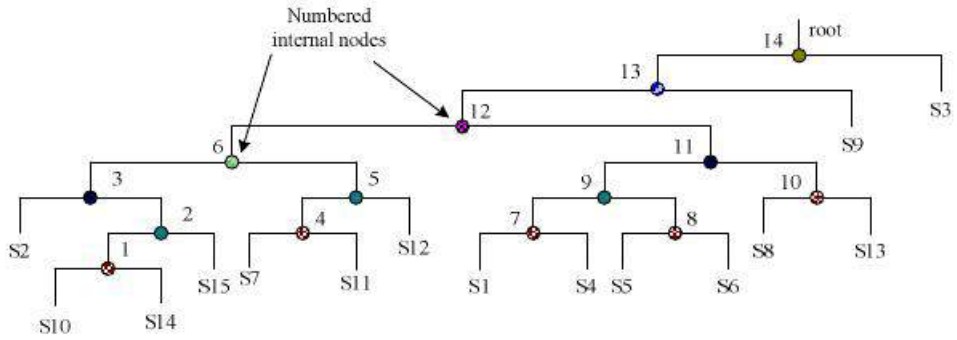


Figura 5.4: Exemplo de uma árvore enraizada produzida pelo método NJ, onde os índices representam a ordem em que as sequências serão alinhadas.

5.2.3 Alinhamento Progressivo

Como foi dito na seção 5.1.3, os alinhamentos da árvore guia que estão no mesmo nível podem ser feitos em paralelo.

Inicialmente, usando o método de busca em profundidade, em pós-ordem, os nós internos são numerados e suas dependências nas sub-árvores esquerda e direita são calculadas. Cada nó é armazenado em três vetores, que armazenam os filhos da sub-árvore direita, os filhos da sub-árvore esquerda e

um vetor com *flags* para indicar quando o alinhamento de algum dos filhos foi feito.

O alinhamento progressivo é feito iterativamente com vários passos. Primeiro, todos os alinhamentos que podem ser feitos nessa iteração são identificados e setados no vetor de *flags*. Se os filhos desse alinhamento já foram calculados, então o alinhamento pode prosseguir para o passo seguinte. No segundo passo, os alinhamentos são feitos paralelamente na GPU e os espaços são inseridos nas sequências. Finalmente, todos os alinhamentos feitos têm seu valor atualizado no vetor de *flags*.

Capítulo 6

Implementação

Neste capítulo, descreveremos os métodos utilizados para a paralelização híbrida do Clustal W.

6.1 Matriz de Distâncias

A divisão das sequências entre os processadores é o mesmo método utilizado em [MSMW] e consiste em dividir as sequências em p conjuntos, onde p é o número de processadores. Cada processador faz o alinhamento, utilizando as GPUs, dessas k/p sequências e as envia para os outros processadores, fazendo com que cada processador tenha uma matriz $(k/p) \times k$.

Para o cálculo da matriz de distâncias na GPU, utilizamos duas matrizes auxiliares para armazenar os alinhamentos parciais contendo *gaps* na vertical e *gaps* na horizontal, respectivamente, a matriz E e a matriz F [BFK+11]. Com isso, para $0 \leq i \leq n$ e $0 \leq j \leq m$, teremos:

$$A[i,j] = \max \begin{cases} E[i,j] \\ F[i,j] \\ A[i-1,j-1] + c(s[i],t[j]) \end{cases} \tag{6.1}$$

$$E[i,j] = \max \begin{cases} E[i,j-1] - G_{ext} \\ E[i,j-1] - G_{open} \end{cases} \tag{6.2}$$

$$F[i,j] = \min \begin{cases} F[i-1,j] - G_{ext} \\ F[i-1,j] - G_{open} \end{cases} \tag{6.3}$$

Onde, G_{open} é o custo por inserir um *gap* na sequência e G_{ext} é o custo por estender esse *gap*, ou seja, adicionar outro *gap* no alinhamento. A inicialização da matriz A pode ser expressa pelas Equações 6.4 e 6.5:

$$A[i, 0] = -i \ . \ G_{ext} \ - \ G_{open} \tag{6.4}$$

$$A[0, j] = -j \ . \ G_{ext} \ - \ G_{open} \tag{6.5}$$

As matrizes E e F são inicializadas com o valor $-\infty$ na primeira linha e na primeira coluna de ambas as matrizes. Inicializadas as matrizes A, E e F, podemos continuar com a execução do algoritmo porém, como o alinhamento de pares de sequências feito em vários pares simultaneamente consome muita memória, dividimos esse problema em subproblemas. Estes subproblemas, chamados de *windows*, são conjuntos de alinhamentos de pares de sequências, onde estas últimas foram ordenadas em ordem decrescente pelo tamanho da sequência. Por sua vez, podemos considerar cada *window* como um *grid* contendo um número constante de blocos.

6.2 Árvore Guia

Como foi dito na Seção 3.3.2, os métodos baseados em distâncias tem uma complexidade menor e tem mais versões que executam em paralelo [GG03], tanto em granulosidade fina quanto em granulosidade grossa, enquanto que os métodos baseados em evolução mínima não possuem muitas versões que executam em paralelo.

Na implementação proposta, o método utilizado para criar a árvore guia é o *Windjammer* [MSMW], que é uma implementação utilizando MPI do *NINJA*, onde a matriz de distâncias D é dividida e cada processador recebe k/p linhas dessa matriz, sendo k o número de sequências e p é o número de processadores. Para facilitar a comunicação, os processadores armazenam a linha inteira, mesmo que somente uma porção dela seja utilizada. Além disso, cada D_{ij} deve ser armazenado em um compartimento, que também são distribuídos entre os processadores, fazendo com que cada D_{ij} seja armazenado duas vezes e com que quase toda a matriz D seja armazenada por pelo menos dois processadores diferentes. Com a matriz D distribuída, os processadores calculam, utilizando as GPUs, o valor para R_i e o valor R_x^{max} é divido entre eles de tal forma que o número de itens em cada compartimento seja o mais

igual possível. O pseudo-código paralelo para o *Windjammer* pode ser visto no Algoritmo 4.

Algoritmo 4 Windjammer

1: Construa os compartimentos
2: **para** K = nSequences **faça**
3: Utilizando as GPUs, encontre os mínimos locais minI e minJ
4: Encontre os mínimos globais para minI e minJ
5: Espalhe as linhas que contêm os mínimos globais
6: Crie a nova linha na GPU
7: Insira o novo d na GPU ou reconstrua seus compartimentos
8: Estime o novo minS
9: **fim para**

Para cada iteração, as equações da Seção 3.3.2 são utilizadas nas GPUs para calcular o mínimo local de cada processo na linha 2 do algoritmo. O próximo passo é encontrar o mínimo global e, com os mínimos globais encontrados, os processadores que possuem as linhas dos mínimos espalham para os demais essas linhas. No próximo passo, uma nova linha é criada pela junção das *OTU's* i e j. Com o passar das iterações e das remoções de linhas e colunas da matriz, a soma das colunas R_i vai diminuindo em um ritmo desbalanceado entre os processos. Isto faz que os compartimentos podem se sobrepor, o que implica numa perda de eficiência no primeiro filtro. Para evitar isso, os compartimentos são refeitos quando o número de *OTU's* tiver sido reduzido pela metade.

6.3 Alinhamento Progressivo

O método para alinhamento utilizado na Seção 6.1 também é utilizado para fazer o alinhamento entre pares de sequências nas GPUs nesse Estágio e uma modificação do mesmo é utilizada para fazer o alinhamento entre perfis. Para essa modificação, a inicialização da matriz A pode ser expressa pelas Equações 6.6 e 6.7, como visto em [LLM10].

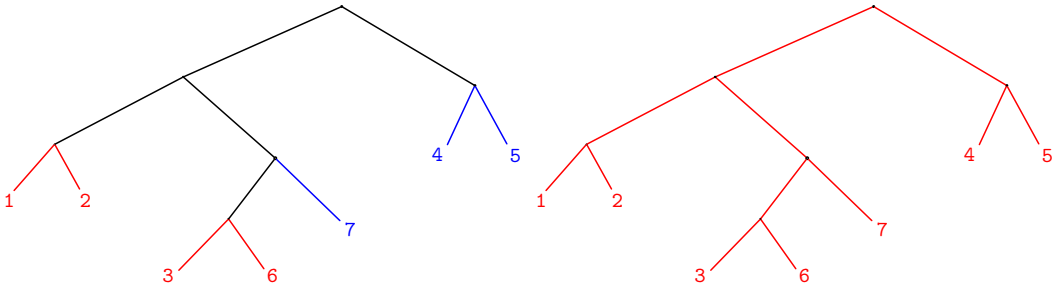
$$A[i,0] = G\ .\ Ns_{col}\ .\ (Ns_{row} - NG_{row}(i)) + A[0,i-1] \tag{6.6}$$

$$A[0,j] = G\ .\ Ns_{row}\ .\ (Ns_{col} - NG_{col}(j)) + A[j-1,0] \tag{6.7}$$

Onde, G é o custo de se abrir um *gap*, Ns é o número de elementos na linha(*row*) ou na coluna(*col*) do perfil e $NG(i)$ é o número de *gaps* na i-ésima posição do perfil. O cálculo das próximas posições da matriz é feito utilizando a Equação 6.8, onde o custo $SP_C(i,j)$ é a pontuação por soma de pares descrita na Seção 3.1.

$$A[i,j] = \max \begin{cases} A[i,j-1] + G \ . \ Ns_{row} \\ A[i-1,j] + G \ . \ Ns_{col} \\ A[i-1,j-1] + SP_C(i,j) \end{cases} \tag{6.8}$$

Para a distribuição das sequências a serem alinhadas, dois métodos foram desenvolvidos e testados. No primeiro, utilizando uma distribuição por sub-árvore, realizando $\log(p)$ passos de comunicação. Nesse caso, o número de processadores diminui pela metade conforme a árvore vai sendo computada, sobrando apenas um único processador ao final da árvore, o que pode ser visto na Figura 6.1.



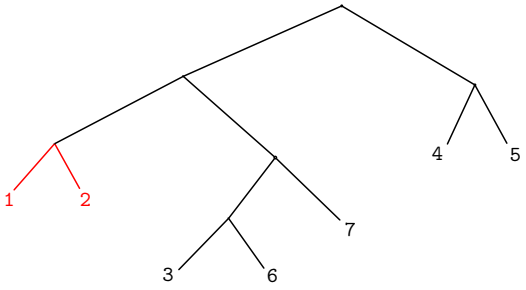
(a) Primeiro passo: Os alinhamentos são divididos entre os processadores.

(b) Ao final do primeiro passo, metade dos processadores envia suas sequências alinhadas para a outra metade dos processadores.

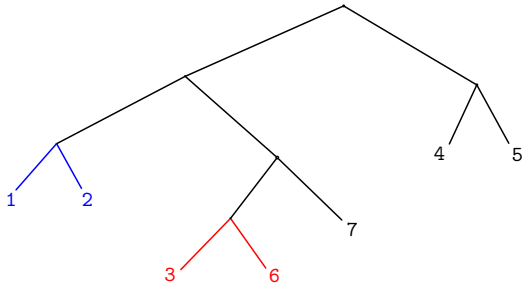
Figura 6.1: Construção do alinhamento progressivo utilizando o segundo método com 2 processadores.

E no segundo, utilizando uma distribuição do tipo mestre-escravo, onde os escravos recebem um par de sequências, ou de perfis, conforme os pares de sequências, ou de perfis, vão surgindo na árvore, o que pode ser visto na Figura 6.2.

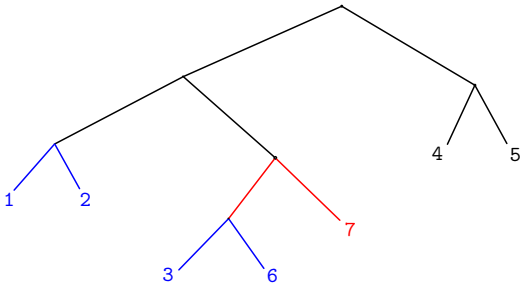
Entretanto, devido ao tempo gasto com a execução do segundo método ser muito superior ao do primeiro método, este se torna inviável.



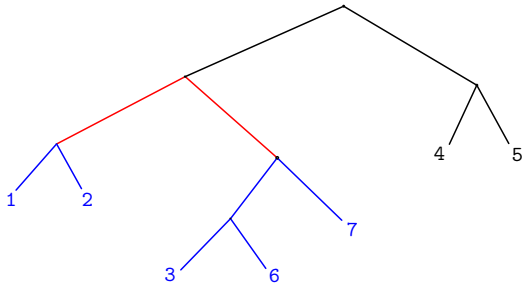
(a) Junção das *OTU's* 5 e 6.



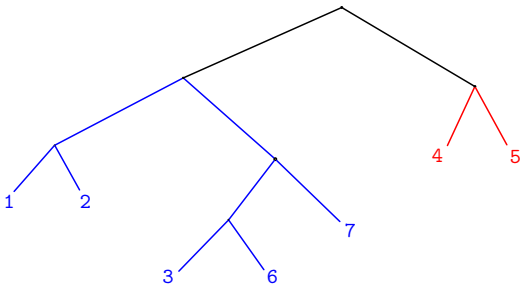
(b) Árvore com junção das *OTU's* 5-6 e 4.



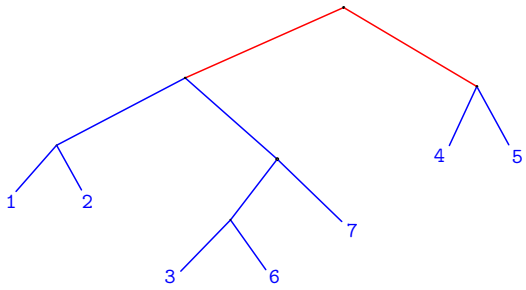
(c) Árvore com junção das *OTU's* 4-5-6 e 7.



(d) Junção das *OTU's* 1 e 2.



(e) Árvore guia final.



(f) Árvore guia final.

Figura 6.2: Construção do alinhamento progressivo utilizando o primeiro método com 2 processadores.

Capítulo 7

Resultados

Neste capítulo apresentaremos os casos de teste e os resultados obtidos pela implementação proposta e pelo *ClustalW-MPI*. A medida de tempo para alinhar várias sequências foi utilizada para determinar o desempenho dos métodos, onde foram feitos 6 casos de teste extraídos da base de dados HIV-1, disponibilizado em [ncb], como foi feito em [LSM09]:

Caso 01: 400 sequências de tamanho médio 851;

Caso 02: 1000 sequências de tamanho médio 851;

Caso 03: 2000 sequências de tamanho médio 234;

Caso 04: 4000 sequências de tamanho médio 238;

Caso 05: 4000 sequências de tamanho médio 61; e

Caso 06: 8000 sequências de tamanho médio 68.

7.1 Ambiente de testes

Os casos de teste foram executados em um *cluster* contendo 40 nós com processadores Intel(R) Xeon(R) CPU X3440 @2.53GHz de 8 núcleos, 4 Gb de memória RAM, sistema operacional Rocks 6.2 e a comunicação entre eles é feita através de placas Myrinet-10G, sendo 8 destes computadores foram equipados com 1 GPU Nvidia Quadro 600 cada. Cada GPU possui 1 Gb de memória com 2 *Streaming Multiprocessors (SMs)* e 48 *Streaming Processors (SPs)* em cada SM, totalizando 96 CUDA cores.

7.2 Desempenho dos métodos

Na Tabela 7.1, vemos as medidas de tempo das execuções serial e paralelas do *ClustalW-MPI* para os casos de teste estabelecidos, onde podemos ver que o tempo de execução é proporcional ao número de sequências e ao tamanho das sequências que vão ser alinhadas, o que também pode ser visto na Figura 7.1.

Caso de teste		Número de nós				
	Serial	02	04	08	16	32
Caso 01	1749	1772	602	259	121	67
Caso 02	11043	11116	3751	1622	767	379
Caso 03	2352	2506	937	485	296	224
Caso 04	10373	11318	4501	2551	1707	1387
Caso 05	904	1738	1290	1168	1051	1049
Caso 06	6072	12605	9826	9202	8364	8388

Tabela 7.1: Resultados obtidos, em segundos, utilizando o *ClustalW-MPI*.

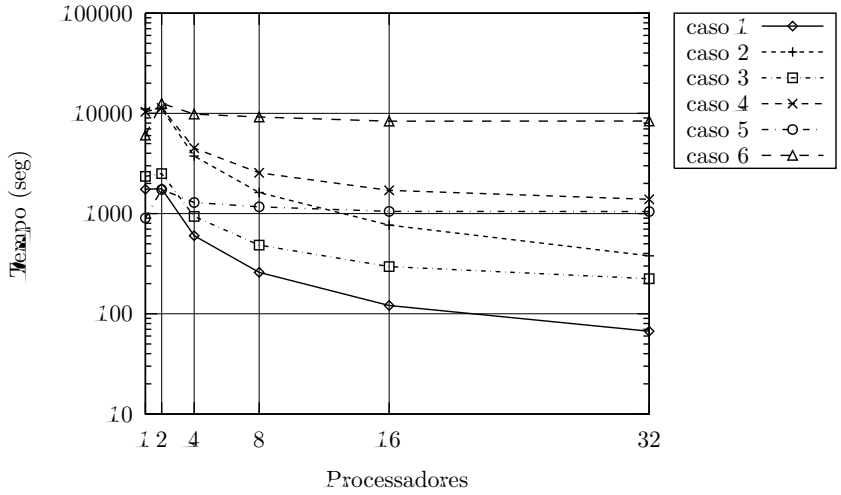


Figura 7.1: Tempo de execução do *ClustalW-MPI*.

Fazendo uma comparação entre a Tabela 7.2 e a Tabela 7.1, podemos ver que, com apenas 1 nó, os tempos de execução da paralelização híbrida são bem menores. A paralelização do Estágio 1 do algoritmo na GPU tem uma

grande redução no tempo de execução, o que faz o tempo total de execução ficar baixo pois, o Estágio 1 é o estágio que mais demora para ser executado. A implementação utilizada para a construção da árvore guia, no Estágio 2, também fez com que o tempo de execução permanecesse baixo.

Caso de teste	Número de nós			
	01	02	04	08
Caso 1	0.39	0.41	0.48	0.6
Caso 2	1.34	1.3	1.96	2.35
Caso 3	5.07	3.32	2.62	2.61
Caso 4	17.39	12.3	9.07	9.95
Caso 5	14.78	10.03	9.11	8.95
Caso 6	68.85	45.37	45.49	48.17

Tabela 7.2: Resultados obtidos, em segundos, utilizando a implementação proposta.

A expressão gráfica da Tabela 7.2 pode ser vista na Figura 7.2, onde podemos ver que, para os casos de teste com poucas sequências de tamanho grande, a paralelização híbrida não foi eficiente e, a paralelização utilizando uma GPU, obteve o mesmo desempenho da paralelização utilizando 8 GPUs.

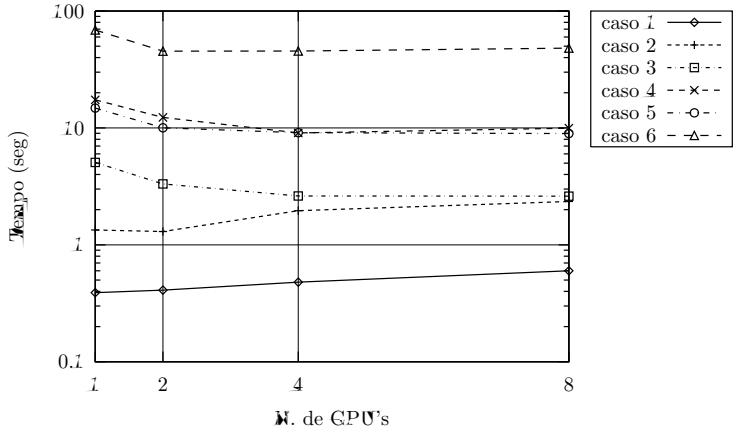


Figura 7.2: Tempo de execução da implementação proposta.

Como é visto na literatura, o primeiro estágio do algoritmo do ClustalW é o estágio que pode ser melhor paralelizado, o que é evidenciado pela Figura 7.3, onde temos o tempo de execução do Estágio 1 para os casos de teste propostos com 1, 2, 4 e 8 GPUs.

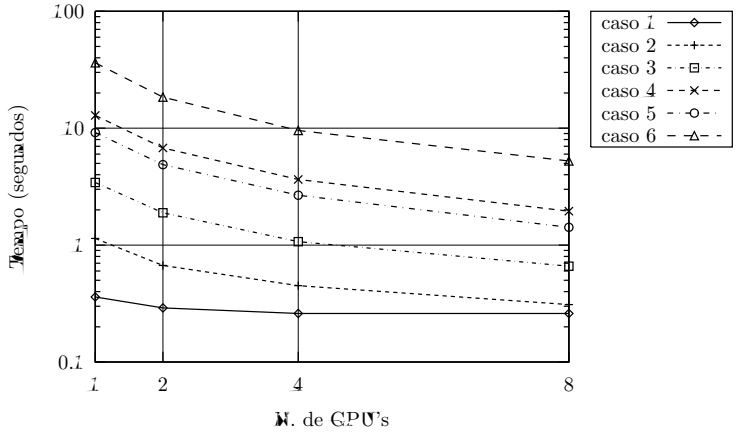


Figura 7.3: Execução do Estágio 1.

Diferente do Estágio 1, o Estágio 2 não obteve um comportamento decrescente com relação ao tempo de execução conforme adicionamos mais GPUs. Isto se deve à comunicação necessária para se completar cada ciclo do algoritmo NJ. A Figura 7.4, mostra o tempo de execução do Estágio 2 para os casos propostos com 1, 2, 4 e 8 GPUs, onde podemos observar que a paralelização deste estágio com várias GPUs ganha da paralelização com uma GPU na maioria dos casos e perde quando temos que alinhar poucas sequências de tamanho grande.

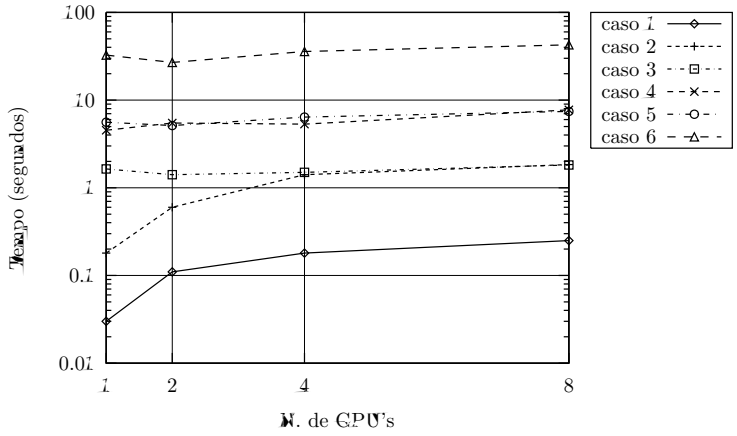


Figura 7.4: Execução do Estágio 2.

Na Figura 7.5, temos o tempo de execução do Estágio 3 para os casos propostos com 1, 2, 4 e 8 GPUs. Nesse estágio, os tempos ficaram na casa dos milésimos de segundo para todas as paralelizações, o que acaba por não influenciar no desempenho geral do algoritmo em alguns casos. Entretanto, a execução Estágio 3 não passou da casa dos décimos de segundo em nenhuma execução do algoritmo.

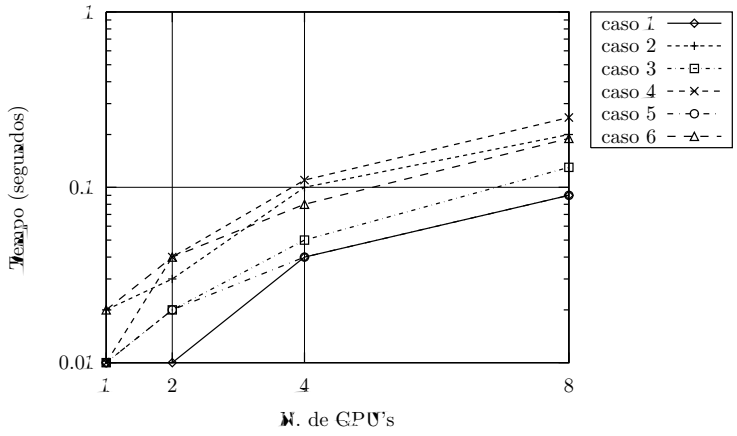
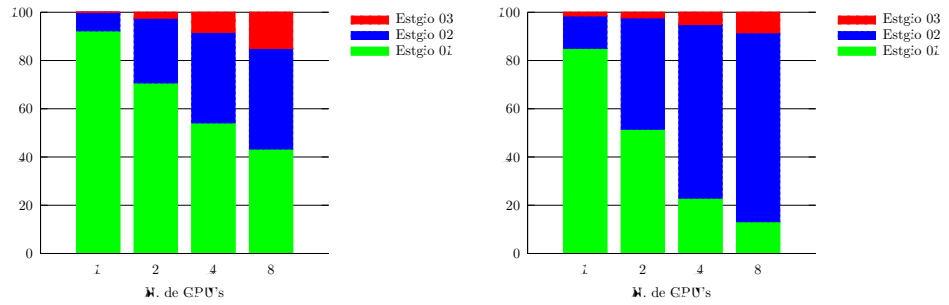


Figura 7.5: Execução do Estágio 3.

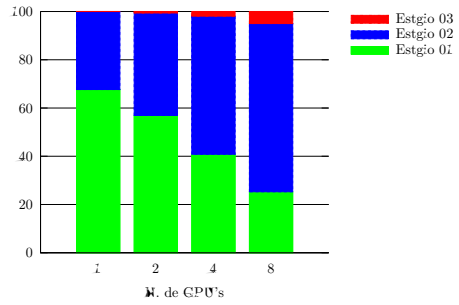
Como podemos ver na Figura 7.6a, o tempo utilizado pelo Estágio 1 do método é o que necessita de maior tempo de execução quando temos poucas sequências e o tamanho delas é grande, o mesmo não é observado quando

temos muitas sequências pequenas. Nas Figuras de 7.6b até 7.6f, o tempo de execução do Estágio 2 passa a ser maior conforme aumentamos o número de sequências a serem alinhadas.

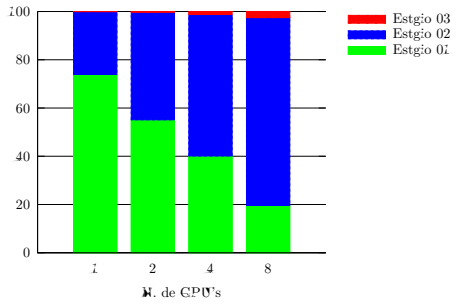


(a) Execução do caso 01.

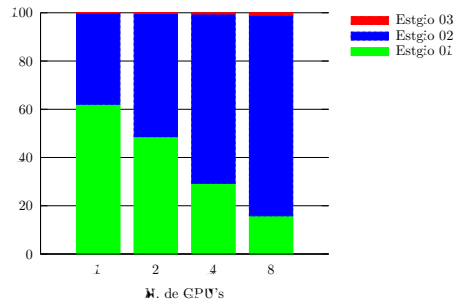
(b) Execução do caso 02.



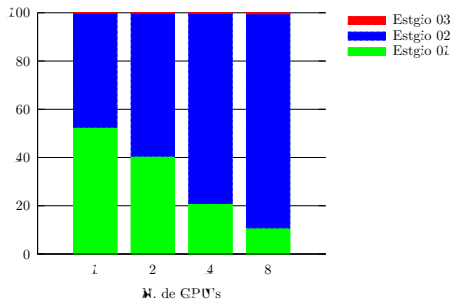
(c) Execução do caso 03.



(d) Execução do caso 04.



(e) Execução do caso 05.



(f) Execução do caso 06.

Figura 7.6: Porcentagem de tempo gasto com cada Estágio.

Na Tabela 7.3, temos o *speedup* obtido, em relação à execução sequencial do algoritmo *ClustalW*, pela implementação proposta para cada caso de teste, o que também pode ser visto na Figura 7.7. Comparando com a execução sequencial obtivemos *speedups* que ficaram entre 8200, no caso onde temos um número pequeno de sequências de tamanho grande, e 61, no caso onde temos uma grande quantidade de sequências de tamanho pequeno.

Caso de teste	Número de nós			
	01	02	04	08
Case 1	4484.62	4265.85	3643.75	2915.00
Case 2	8241.04	8494.62	5634.18	4699.15
Case 3	463.91	708.43	897.71	901.15
Case 4	596.49	843.33	1143.66	1042.51
Case 5	61.16	90.13	99.23	101.01
Case 6	88.19	133.83	133.48	126.05

Tabela 7.3: *Speedup* obtido em relação à execução sequencial do algoritmo *ClustalW*.

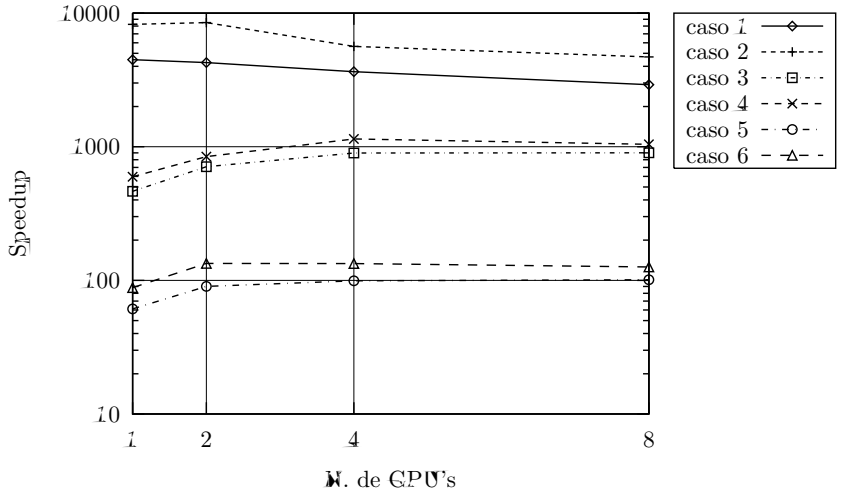


Figura 7.7: *Speedup* obtido em relação à execução sequencial do algoritmo *ClustalW*.

Na Tabela 7.4, temos o *speedup* obtido, em relação à execução paralela utilizando 32 computadores do algoritmo *ClustalW-MPI*, pela implementação

proposta para cada caso de teste, o que também pode ser visto na Figura 7.8. Nessa comparação, podemos ver que a implementação proposta tem um desempenho melhor do que a do *ClustalW-MPI*, utilizando 32 computadores, obtendo *speedups* que ficaram entre 44, em um caso intermediário de número de sequências de tamanho médio, e 290, em um caso com um número menor de sequências de tamanho grande.

De uma forma geral, o *speedup* é maior quando temos poucas sequências de tamanho grande para serem alinhadas e menor quando temos muitas sequências de tamanho pequeno. A comunicação entre os processadores e a complexidade dos Estágios 2 e 3 também fizeram com que o *speedup* obtido não fosse o ideal para um número crescente de GPUs.

Caso de teste	Número de nós			
	01	02	04	08
Case 1	171.79	163.41	139.58	111.67
Case 2	282.84	291.54	193.37	161.28
Case 3	44.18	67.47	85.50	85.82
Case 4	79.76	112.76	152.92	139.40
Case 5	70.97	104.59	115.15	117.21
Case 6	121.83	184.88	184.39	174.13

Tabela 7.4: *Speedup* obtido em relação à execução utilizando 32 computadores do algoritmo *ClustalW-MPI*.

Também foi feito um estudo utilizando uma segunda abordagem para a distribuição de sequências no Estágio 3 e os resultados dessa abordagem podem ser vistos na Figura 7.9. Essa nova distribuição, do tipo mestre-escravo, não é eficaz e perde para a distribuição anterior, o que podemos na Figura 7.10.

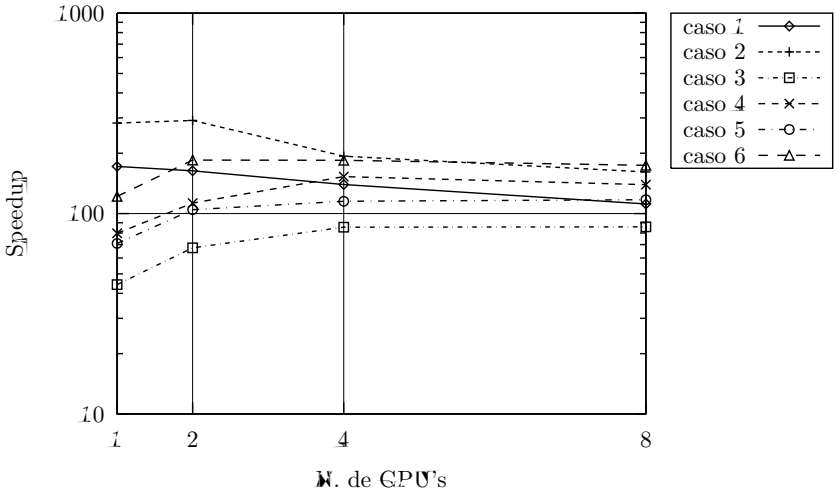


Figura 7.8: *Speedup* obtido em relação à execução utilizando 32 computadores do algoritmo *ClustalW-MPI*.

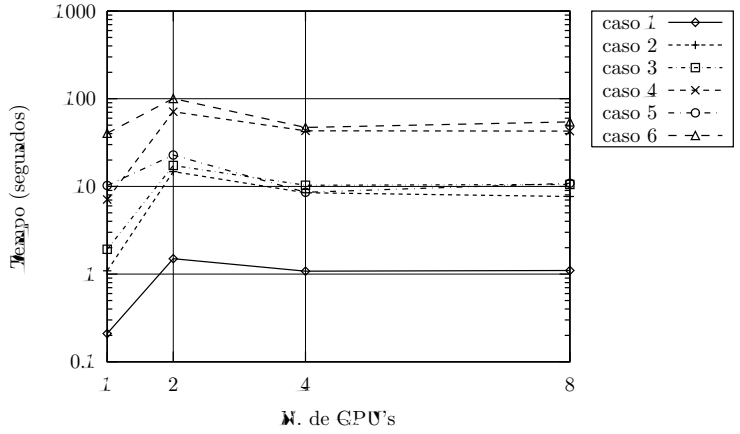


Figura 7.9: Execução do Estágio 3 utilizando a distribuição do tipo mestre-escravo.

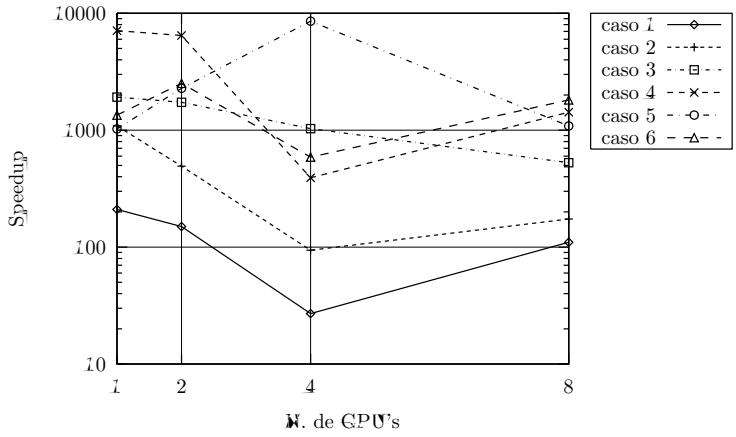


Figura 7.10: *Speedup* da implementações do Estágio 3.

Capítulo 8

Conclusão

A utilização de técnicas de alinhamento é importante para se conhecer novos membros de uma determinada família, sua história evolutiva bem como no processo de sequenciamento de DNA. Entretanto, o alinhamento de várias sequências é um problema NP-difícil, o que impossibilita a sua execução serial para um número grande de sequências. Por esse motivo, métodos utilizando programação paralela foram desenvolvidos. E, com o aparecimento de novos ambientes de programação paralela, esses métodos foram otimizados e houve um grande ganho de desempenho no alinhamento de várias sequências.

Muitos métodos para alinhamento de várias sequências foram desenvolvidos visando esses novos ambientes de programação, como o *ClustalW-MPI* e o MSA-CUDA. Neste trabalho, apresentamos uma implementação para o *ClustalW* utilizando MPI e CUDA, visando um algoritmo híbrido no qual todos os estágios são paralelizados.

Os *speedups* obtidos ficaram entre 61 e 8200 e entre 44 e 280, quando comparados com a execução sequencial do *ClustalW* e com a versão paralela utilizando 32 computadores do *ClustalW-MPI*, respectivamente. A paralelização do Estágio 1 utilizando CUDA é constantemente abordada na literatura. Entretanto, o mesmo não acontece com os Estágios 2 e 3, que são mais difíceis de serem paralelizados, devido à grande dependência de dados envolvida em cada Estágio e à complexidade dos métodos.

Referências Bibliográficas

[BFK+11] Blazewicz, Jacek, Wojciech Frohmberg, Michal Kierzynka, Erwin Pesch e Pawel Wojciechowski: *Protein alignment algorithms with an efficient backtracking routine on multiple GPUs.* BMC Bioinformatics, 12:181, 2011.

[CKT06] Chaichoompu, K., S. Kittitornkun e S. Tongsima: *MT-ClustalW: multithreading multiple sequence alignment.* Em *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006.

[CLRS09] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest e Clifford Stein: *Introduction to Algorithms, Third Edition.* The MIT Press, 2009.

[dB03] Brito, Rogério Theodoro de: *Alinhamento de Sequências Biológicas.* Tese de Mestrado, Instituto de Matemática e Estatística - Universidade de São Paulo, 2003.

[DE06] Datta, Amitava e Justin Ebedes: *Multiple Sequence Alignment in Parallel on a Cluster of Workstations*, capítulo 8. John Wiley & Sons, Inc, 2006.

[DFRC93] Dehne, Frank K. H. A., Andreas Fabri e Andrew Rau-Chaplin: *Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers.* Em *Symposium on Computational Geometry*, página 298–307, 1993. `http://dblp.uni-trier.de/db/conf/compgeom/compgeom93.html#DehneFR93`.

[dS08] Santos, Danielle Furtado dos: *Alinhamento Múltiplo de Proteínas via Algoritmo Genético Baseado em Tipos Abstratos de*

Dados. Tese de Mestrado, Instituto de Computação da Universidade Federal de Alagoas, 2008.

[Edg04] Edgar, Robert: *MUSCLE: a multiple sequence alignment method with reduced time and space complexity*. BMC Bioinformatics, 5:113, 2004.

[FW78] Fortune, Steven e James Wyllie: *Parallelism in random access machines*. Em *Proceedings of the tenth annual ACM symposium on Theory of computing*, 1978.

[GFB+04] Gabriel, Edgar, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham e Timothy S. Woodall: *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. Em *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[GG03] Guindon, S. e O. Gascuel: *A simple, fast and accurate algorithm to estimate large phylogenies by maximum likelihood*. Systematic Biology, 52:696–704, 2003. `http://www.bibsonomy.org/bibtex/2a6f4462ddab20318a54d4fe69d919e1e/stephane.guindon`.

[Gla09] Glaskowsky, Peter N.: *NVIDIA's Fermi: The First Complete GPU Computing Architecture*. Relatório Técnico, NVIDIA Corporation, 2009.

[HBD02] Howe, Kevin, Alex Bateman e Richard Durbin: *QuickTree: building huge Neighbour-Joining trees of protein sequences.* Bioinformatics, 18(11):1546–7, novembro 2002.

[LLM10] Lopes, Heitor S., Carlos R. Erig Lima e Guilherme L. Moritz: *A Parallel Algorithm for Large-Scale Multiple Sequence Alignment*. Compunting and Informatics, 29:1233–1250, 2010.

[Lou10] Loureiro, Leonardo Vinícius Rolan: *Algoritmos BSP/CGM para Programação Dinâmica*. Tese de Mestrado, Faculdade de Computação - Universidade Federal de Mato Grosso do Sul, 2010.

[LSM09] Liu, Yongchao, B. Schmidt e D.L. Maskell: *MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA*. Em *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, 2009.

[LSM11] Liu, Yongchao, B. Schmidt e D.L. Maskell: *An Ultrafast Scalable Many-Core Motif Discovery Algorithm for Multiple GPUs*. Em *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011.

[LSVMW06] Liu, Weiguo, Bertil Schmidt, Gerrit Voss e Wolfgang Müller-Wittig: *GPU-ClustalW: Using Graphics Hardware to Accelerate Multiple Sequence Alignment*. Em Robert, Yves, Manish Parashar, Ramamurthy Badrinath e Viktor K. Prasanna (editores): *HiPC*, volume 4297 de *Lecture Notes in Computer Science*, página 363–374. Springer, 2006, ISBN 3-540-68039-X.

[MGM+12] Munshi, Aaftab, Benedict R. Gaster, Timothy G. Mattson, James Fung e Dan Ginsburg: *OpenCL Programming Guide*. Pearson, 2012.

[MM88] Myers, Eugene W. e Webb Miller: *Optimal alignments in linear space*. CABIOS, 4:11–17, 1988.

[mpia] `http://www.mcs.anl.gov/research/projects/mpi/`.

[mpib] `http://www.mpich.org/`.

[MSMW] McLay, Robert, Dan Stanzione, Sheldon McKay e Travis Wheeler: *A Scalable Parallel Implementation of the Neighbor Joining Algorithm for Phylogenetic Trees*.

[ncb] `http://www.ncbi.nlm.nih.gov/`.

[NHH00] Notredame, Cédric, Desmond G. Higgins e Jaap Heringa: *T-coffee: a novel method for fast and accurate multiple sequence alignment*. Journal of Molecular Biology, 1:205–217, 2000.

[PDA09] Price, Morgan N., Paramvir S. Dehal e Adam P. Arkin: *FastTree: Computing Large Minimum Evolution Trees with Profiles instead of a Distance Matrix*. Molecular Biology and Evolution, 26(7):1641–1650, 2009. `http://mbe.oxfordjournals.org/content/26/7/1641.abstract`.

[Sil03] Silva, Marco Túlio Nogueira: *Alinhamento Múltiplo Global de Sequências pela Representação de Profile e Clusterização: Comparação com os Resultados do CLUSTAL W (EMBL-EBI)*. Tese de Mestrado, Departamento de Ciência da Computação da Universidade Federal de Lavras, 2003.

[SK88] Studier, J. A. e K. J. Keppler: *A note on the Neighbor-Joining algorithm of Saitou and Nei.* Molecular Biology and Evolution, 5:729–731, 1988.

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

[SMP08] Simonsen, Martin, Thomas Mailund e Christian N. Pedersen: *Rapid Neighbour-Joining*. Em *Proceedings of the 8th international workshop on Algorithms in Bioinformatics*, WABI '08, página 113–122, Berlin, Heidelberg, 2008. Springer-Verlag, ISBN 978-3-540-87360-0.

[SN87] Saitou, Naruya e Masatoshi Nei: *The Neighbor-Joining Method: A New Method for Reconstructing Phylogenetic Trees*. Molecular Biology and Evolution, 4:406–425, 1987.

[SSKLK13] Shreiner, Dave, Graham Sellers, John Kessenich e Bill Licea-Kane: *OpenGL Programming Guide*. Pearson, 2013.

[Ste03] Stefanes, Marco Aurélio: *Algoritmos Paralelos de Granulosidade Grossa em Grafos Bipartidos Convexos*. Tese de Doutoramento, Instituto de Matemática e Estatística - Universidade de São Paulo, 2003.

[THG94] Thompson, Julie D., Desmond G. Higgins e Toby J. Gibson: *CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-*

specific gap penalties and weight matrix choice. Nucleic Acids Research, 22:4673–4680, 1994.

[Val90] Valiant, Leslie G.: *A Bridging Model for Parallel Computation.* Commun. ACM, 33(8):103–111, 1990.

[Whe09] Wheeler, Travis J.: *Large-scale neighbor-joining with NINJA*. Em *Proceedings of the 9th international conference on Algorithms in bioinformatics*, WABI'09, página 375–389, Berlin, Heidelberg, 2009. Springer-Verlag, ISBN 3-642-04240-6, 978-3-642-04240-9.