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

Estratégias paralelas para alinhamento de sequências biológicas em espaço linear com algoritmo Myers-Miller em CPU

Eduardo Monteiro de Castro Gomes

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientadora
Prof.a Alba Cristina M. A. de Melo

Brasília
2017

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

Estratégias paralelas para alinhamento de sequências biológicas em espaço linear com algoritmo Myers-Miller em CPU

Eduardo Monteiro de Castro Gomes

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof.a Alba Cristina M. A. de Melo (Orientadora)
CIC/UnB

Prof.a Aleteia Patricia Favacho de Araujo
CIC/UnB

Prof. Eduardo Adilo Pelinson Alchieri
CIC/UnB

Prof. Pedro Antonio Dourado Rezende
Coordenador do Curso de Computação — Licenciatura

Brasília, 11 de Dezembro de 2017

Sumário

1 Introdução **1**

2 Revisão de Literatura **4**

2.1 Conceitos básicos 4

2.1.1 Alinhamento de Sequências 4

2.1.2 *Score* 5

2.2 Algoritmos para alinhamento 6

2.2.1 Algoritmo Needleman-Wunsch (NW) 6

2.2.2 Algoritmo Smith-Waterman (SW) 8

2.2.3 Algoritmo Hirschberg 9

2.2.4 Algoritmo de Gotoh 11

2.2.5 Algoritmo de Myers e Miller (MM) 12

3 Projeto de Implementações Paralelas em CPU **14**

3.1 Implementações do Algoritmo Myers e Miller 14

3.2 Experimentos 18

4 Resultados Experimentais **20**

4.1 Determinação do Limite L para Utilização na Implementação Multi-processos 20

4.2 Avaliação de Diferença entre as Implementações 21

4.3 Avaliação de Ganho de Desempenho pelas Implementações Paralelas 22

5 Conclusões e Trabalhos Futuros **24**

5.1 Conclusões 24

5.2 Trabalhos futuros 25

Referências **26**

Lista de Figuras

2.1 Subdivisão da Matriz de Programação Dinâmica pelo Método de Hirschberg. 10
2.2 Processo para Determinação da Coluna por onde Passa o Alinhamento Ótimo. 11

3.1 Representação do estado da fila em diferentes instantes. 15
3.2 Diagrama com as subdivisões consideradas nas implementações do algoritmo de Myers e Miller. 16

4.1 Distribuições dos tempos para alinhamento de sequências de tamanho 10000 para diferentes valores de L 20
4.2 Distribuições dos tempos para alinhamento de sequências de tamanho 10000 22
4.3 Avaliação do ganho de performance *speedup* pelo uso de paralelização 23

Lista de Tabelas

2.1 Exemplo de sequências 4
2.2 Exemplos de Alinhamento 5
2.3 Matriz de programação dinâmica para alinhamento global 7
2.4 Matriz de programação dinâmica para alinhamento local 9
2.5 Exemplo de Alinhamento Local 9

4.1 Médias e (Desvios-padrões) dos tempos, em segundos, para alinhamento de sequências de tamanho 10000 21
4.2 Médias e (Desvios-padrões) dos tempos, em segundos, para alinhamento de sequências de tamanhos variáveis 23

Capítulo 1

Introdução

Bioinformática pode ser definida como uma disciplina envolvida na aplicação de técnicas computacionais para organizar e compreender diversos aspectos da biologia [1]. Com sua evolução impulsionada pelos avanços tecnológicos, as técnicas de Bioinformática reunem métodos matemáticos e estatísticos para modelar e investigar os princípios organizacionais e a significância dos dados biológicos.

A compreensão sobre as funções específicas dos genes e sobre estruturas moleculares estão entre os principais temas estudados em Bioinformática [1]. A partir da disponibilização de um grande volume de dados de sequenciamentos genéticos, o estudo sobre expressão genética tem como problema fundamental o alinhamento entre sequências. Por meio do alinhamento entre duas sequências pode-se explorar evidências de relacionamento evolutivo ou funcional entre diferentes organismos.

Neste trabalho são considerados os alinhamentos do tipo global entre pares de sequências biológicas. Para esse tipo de alinhamento, busca-se a maior sub-sequência comum entre as sequências, permitindo inserções e deleções e fazendo com que o alinhamento se extenda para todo o comprimento das sequências.

De uma forma geral, pode-se considerar que o problema em Bioinformática de alinhamento entre duas sequências biológicas não é diferente do problema em ciência de computação de comparação de *strings*, e os métodos para tais problemas podem ser utilizados ou adaptados.

Para avaliar os diferentes possíveis alinhamentos entre duas sequências utiliza-se uma função *score* para representar as semelhanças ou diferenças entre as sequências. A estrutura da função *score* pode ser simplificada ou elaborada para representar as histórias evolucionárias e estruturas multidimensionais das moléculas biológicas [2].

A partir da determinação da estrutura da função *score* e seus pesos é necessária a utilização de técnicas computacionais para obtenção de um alinhamento entre as sequências que otimize a função *score*.

Um dos algoritmos mais tradicionais e que pode ser considerado como ponto de partida neste trabalho é o algoritmo de Needleman-Wunsch [3] que utiliza técnica de programação dinâmica para obtenção de alinhamento ótimo a partir de sua representação matricial. Esse algoritmo exato converge para um alinhamento ótimo e tem complexidade de tempo e espaço de memória da ordem $\mathcal{O}(n^2)$, de forma que essa complexidade quadrática no espaço pode tornar-se impraticável para grandes sequências.

O algoritmo de Hirshberg [4] foi proposto como uma versão eficiente no espaço do algoritmo de Needleman-Wunsch. É baseado na estratégia *Divide and conquer* para subdividir o problema de alinhamento em subproblemas menores de forma recursiva até que os problemas possam ser resolvidos de forma simples. É mais eficiente em memória pois tem complexidade linear $\mathcal{O}(n)$ no espaço mas preserva a complexidade da ordem $\mathcal{O}(n^2)$ no tempo. As estruturas da função *score* consideradas nesses algoritmos são bastante simplificadas e foram aperfeiçoadas em publicações posteriores.

A publicação de Gotoh [5] apresentou o modelo *afine-gap* como uma evolução para estrutura da função *score*. Nessa versão diferentes pesos são associados ao se iniciar ou extender uma série de *gaps* consecutivos em um alinhamento. Esta formulação da função permite captar de forma mais adequada estruturas de relações evolutivas entre duas sequências. A utilização desse modelo de *score* foi apresentado em sua publicação a partir de funções de recorrência com complexidade de tempo e espaço da ordem $\mathcal{O}(n^2)$.

Com proposta análoga à de Hirshberg [4], o algoritmo de Myers-Miller [6] adapta o algoritmo proposto por Gotoh [5], usando a estratégia de subdividir os problemas de alinhamento para atingir complexidade linear no espaço. Esse algoritmo possibilita, portanto, a obtenção de alinhamentos globais ótimos com o modelo *afine-gap* tendo complexidade linear no espaço e quadrática no tempo.

Considerando o grande e crescente volume de dados biológicos disponíveis, e a necessídade em Bioinformática de alinhar e comparar longas sequências, pode-se perceber a demanda por métodos computacionais eficientes que perimitam fazer esses alinhamentos entre milhares de pares de sequências longas nos menores tempos possíveis.

Nas últimas duas décadas foram propostas diferentes alternativas de soluções em computação de alto desempenho para o problema de alinhamento global de sequências. O trabalho de Sandes, Boukerche e Melo [7] apresenta uma classificação para diferentes soluções encontradas na literatura. As diferentes categorias são referentes aos tipos de sequências, tipos de alinhamentos, tipo de função *score*, algoritmo utilizado, complexidade em espaço de memória, tipo e número de unidades de processamento envolvidas.

Nesse artigo, pode ser visto que poucas abordagens na literatura implementam o algoritmo de Myers-Miller e que todas as estratégias listadas que implementam esse algoritmo possuem como alvo GPU (*Graphics Processing Units*) ou o acelerador CellBE, usando

técnicas espeíficas para essas arquiteturas. Após extensa pesquisa não foram encontradas implementações do algoritmo de Myers-Miller para CPU.

Portanto, como alternativa às soluções apresentadas em [7], o presente trabalho tem como objetivo propor técnicas de programação paralela em CPU para a implementação do algoritmo de Myers-Miller, com foco no desempenho do algoritmo em relação ao tempo para alinhamento global de sequencias de DNA usando modelo *affine-gap*.

A proposta de programação paralela é considerada sob duas abordagens, a primeira utilizando múltiplos processos e a segunda múltiplas *threads*. A abordagem de múltiplos processos é na verdade uma abordagem híbrida, na qual a execução paralela é feita até que as subpartições do problema atinjam um limite. Para subpartições menores que esse limite, a execução é sequencial. Ambas as propostas são comparadas a uma implementação sequencial, que também foi implementada no presente trabalho. Experimentos são apresentados e analisados para avaliar o desempenho das diferentes implementações com diferentes pares de sequências de dimensões variadas. Nesses experimentos, são mostradas as vantagens das abordagens paralelas.

Este trabalho está organizado de forma que na Seção 2 é apresentada a revisão da literatura sobre alinhamento de sequências e os algoritmos tradicionais. Na Seção 3 é descrita a proposta para implementação paralela do algoritmo de Myers-Miller e são descritas as formas utilizadas para comparar as diferentes implementações. Os resultados das comparações são apresentados na Seção 4. Na Seção 5 são feitas as conclusões e propostas para trabalhos futuros.

Capítulo 2

Revisão de Literatura

2.1 Conceitos básicos

2.1.1 Alinhamento de Sequências

Alinhamento de sequências é o procedimento para comparar duas ou mais sequências, analisando séries de caracteres individuais ou padrões de caracteres que estão na mesma ordem em ambas as sequências [2].

Dadas duas ou mais sequências, pode-se ter interesse em avaliar alinhamentos de forma global ou local [8]. Quando o interesse está no alinhamento global busca-se a maior sub-sequencia comum entre as duas sequências, permitindo inserções e deleções e fazendo com que o alinhamento se extenda para todo o comprimento das sequências. No alinhamento local busca-se identificar regiões de semelhança dentro das sequências, sem necessáriamente encontrar essas semelhanças ao longo de todo o comprimento das sequências.

Neste trabalho são considerados alinhamentos entre pares de sequências, de forma que as sequências $s = \{s_1, s_2, \ldots, s_m\}$ e $t = \{t_1, t_2, \ldots, t_n\}$ são formadas por caracteres de um alfabeto Σ, conforme exemplo de sequências apresentadas na Tabela 2.1.

Tabela 2.1: Exemplo de sequências

s:	A	T	C	T	T	G	A	C	T	A	A	G	C	A	A	T
t:	T	C	A	G	A	C	T	A	A	T	T	C	A	T	A	

Na busca de alinhamento entre duas sequências é possível inserir espaços '-' em uma ou ambas sequências e esses espaços são chamados de *gaps*. Definem-se s^* e t^* como as sequências extendidas, resultantes da inserção dos espaços. A Tabela 2.2 apresenta dois

exemplos de alinhamentos, $\alpha_1(s,t)$ e $\alpha_2(s,t)$, para as sequências de caracteres apresentadas na Tabela 2.1

A unidade mínima de um alinhamento consiste em uma posição de cada uma das duas sequências sendo comparadas, de forma que podem ser observados os possíveis pares nas comparações: $\{(L,R),(L,-),(-,R)\}$ com $L,R \in \Sigma$. Dentre esses pares, (L,R) representa o alinhamento de um caractere da sequência s^* com um caractere da sequência t^*, podendo esses caracteres serem iguais ou não. Os pares $\{(L,-),(-,R)\}$ representam o alinhamento de um caractere da sequência s^* com um espaço na sequência t^*, e o alinhamento de um espaço na sequência s^* com um caractere da sequência t^* respectivamente.

Alinhamento 1 $\alpha_1(s,t)$																	
s^{*1}:	A	T	C	T	T	G	A	C	T	A	A	-	G	C	A	A	T
t^{*1}:	-	T	C	A	-	G	A	C	T	A	A	T	T	C	A	T	A
Alinhamento 2 $\alpha_2(s,t)$																	
s^{*2}:	A	T	C	T	T	G	A	C	T	A	A	-	G	C	A	A	T
t^{*2}:	-	T	C	-	A	G	A	C	T	A	A	T	T	C	A	T	A

Tabela 2.2: Exemplos de Alinhamento

A partir dessas considerações e do exemplo acima, é simples perceber que são possíveis diversos alinhamentos distintos e por isso é importante determinar se um alinhamento se dá ao acaso ou se as sequências são realmente relacionadas e podem determinar relações biológicas entre os elementos envolvidos [9].

Uma forma de distinguir os diferentes alinhamentos possíveis e avaliar as semelhanças obtidas pelo alinhamento entre as sequências, é por meio da atribuição de um *score* para cada alinhamento. O *score* de um determinado alinhamento é definido a partir de uma regra ou função, de forma que quanto maior o *score* de um determinado alinhamento maior é a semelhança entre as sequências comparadas.

2.1.2 *Score*

Seja α um alinhamento das sequências s e t, e w uma função *score*, então o *score* de α é dado pela soma dos *scores* de cada unidade ou coluna do alinhamento:

$$score(\alpha) = \sum_{i=1}^{|\alpha|} w(s_{[i]}, t_{[i]}). \tag{2.1}$$

e uma possível forma para a função *score* pode considerar por exemplo: $w(x,-) = w(-,x) = -2$, $w(x,y) = 1$ se $x = y$ e $w(x,y) = -1$ se $x \neq y$.

De uma forma geral os métodos de alinhamento buscam encontrar o alinhamento que tenha *score* máximo. O valor máximo de *score* para duas sequências s e t é chamado de similaridade e é definido como:

$$sim(s,t) = \max_{\alpha \in \Lambda(s,t)} score(\alpha), \tag{2.2}$$

em que $\Lambda(s,t)$ é o conjunto de todos os possíveis alinhamentos entre s e t.

Para um par de sequências qualquer, diferentes alinhamentos podem resultar em *scores* iguais, e os diferentes alinhamentos que atingem o *score* máximo, podendo existir um ou mais, são chamados de alinhamentos ótimos $opt(s,t)$.

É importante notar que pode-se representar o alinhamento entre duas sequências a partir de suas semelhanças ou diferenças. No caso de representar as semelhanças, conforme no exemplo e representação acima, busca-se pelo alinhamento que maximiza as semelhanças entre duas sequências. No caso de se representar as diferenças entre as sequências busca-se o alinhamento que minimiza as diferenças. Vale ressaltar que é possível transformar um problema de maximização de semelhanças em um problema de minimização de diferenças a partir de transformações nos pesos considerados na Equação 2.2.

Na seção seguinte são apresentados alguns algoritmos propostos na literatura para obtenção de alinhamentos ótimos.

2.2 Algoritmos para alinhamento

2.2.1 Algoritmo Needleman-Wunsch (NW)

O algoritmo apresentado nesta seção utiliza técnica de programação dinâmica para obtenção de alinhamento ótimo global entre duas sequências. O algoritmo é composto de duas etapas de forma que na primeira etapa é calculada a matriz F *score* de programação dinâmica e na segunda etapa, chamada de *traceback*, é obtido um alinhamento ótimo a partir da matriz calculada.

O cálculo da matriz F é obtido de forma recursiva. Inicia-se pelo preenchimento das bordas superior e esquerda da matriz que representam o alinhamento do prefixo da sequência s com *gaps* na sequência t e alinhamento do prefixo da sequência t com *gaps* na sequência s, respectivamente. Supondo que a penalidade definida na função *score* para inserção de gaps seja descrita por d, define-se que $F(i,0) = -id$ e $F(0,j) = -jd$. Para as demais células da matriz utiliza-se a seguinte função de recursão:

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + w(x_i, y_j), \\ F(i-1, j) + w(x_i, -), \\ F(i, j-1) + w(-, y_j). \end{cases} \tag{2.3}$$

A Tabela 2.3 ilustra a matriz de programação dinâmica calculada para o alinhamento das sequências s e t apresentadas na Tabela 2.1, considerando as seguintes escolhas para a função *score*: *gaps* recebem valor $w(x,-) = w(-,x) = -2$, *matches* recebem valor $w(x,y) = 1$ se $x = y$ e *mismatches* recebem valor $w(x,y) = -1$ se $x \neq y$.

O valor da célula final da matriz $F(m,n)$ é o valor da similaridade para o alinhamento das sequências s e t. Para encontrar o alinhamento ótimo deve-se determinar o caminho na matriz F que resultou no cálculo desse valor de similaridade a partir da função de recorrência da Equação (2.3).

O *traceback* é realizado de forma reversa. A partir da célula final $F(m,n)$ percorre-se a matriz determinando-se de qual célula dentre $(i-1, j-1)$, $(i-1, j)$ ou $(i, j-1)$ foi obtido valor para uma célula $F(i,j)$. Para cada célula do caminho ótimo, a partir da determinação da célula proveniente, adiciona-se um par de símbolos na coluna inícial do alinhamento, de forma que se a célula proveniente foi $(i-1, j-1)$ adiciona-se s_i e t_j, se foi $(i-1, j)$ adiciona-se s_i e um *gap* -", e finalmente caso a célula proveniente seja $(i, j-1)$ adiciona-se um *gap* -"e t_j.

Na Tabela 2.3 os valores em destaque ilustram o caminho que representa o alinhamento ótimo $\alpha_1(s,t)$ exibido na Tabela 2.2.

	-	T	C	A	G	A	C	T	A	A	T	T	C	A	T	A
-	**0**	−2	−4	−6	−8	−10	−12	−14	−16	−18	−20	−22	−24	−26	−28	−30
A	**-2**	−1	−3	−3	−5	−7	−9	−11	−13	−15	−17	−19	−21	−23	−25	−27
T	−4	**-1**	−2	−4	−4	−6	−8	−8	−10	−12	−14	−16	−18	−20	−22	−24
C	−6	−3	**0**	−2	−4	−5	−5	−7	−9	−11	−13	−15	−15	−17	−19	−21
T	−8	−5	−2	**-1**	−3	−5	−6	−4	−6	−8	−10	−12	−14	−16	−16	−18
T	−10	−7	−4	**-3**	−2	−4	−6	−5	−5	−7	−7	−9	−11	−13	−15	−17
G	−12	−9	−6	−5	**-2**	−3	−5	−7	−6	−6	−8	−8	−10	−12	−14	−16
A	−14	−11	−8	−5	−4	**-1**	−3	−5	−6	−5	−7	−9	−9	−9	−11	−13
C	−16	−13	−10	−7	−6	−3	**0**	−2	−4	−6	−6	−8	−8	−10	−10	−12
T	−18	−15	−12	−9	−8	−5	−2	**1**	−1	−3	−5	−5	−7	−9	−9	−11
A	−20	−17	−14	−11	−10	−7	−4	−1	**2**	0	−2	−4	−6	−6	−8	−8
A	−22	−19	−16	−13	−12	−9	−6	−3	0	**3**	1	−1	−3	−5	−7	−7
G	−24	−21	−18	−15	−12	−11	−8	−5	−2	1	**2**	**0**	−2	−4	−6	−8
C	−26	−23	−20	−17	−14	−13	−10	−7	−4	−1	0	1	**1**	−1	−3	−5
A	−28	−25	−22	−19	−16	−13	−12	−9	−6	−3	−2	−1	0	**2**	0	−2
A	−30	−27	−24	−21	−18	−15	−14	−11	−8	−5	−4	−3	−2	1	**1**	1
T	−32	−29	−26	−23	−20	−17	−16	−13	−10	−7	−4	−3	−4	−1	2	**0**

Tabela 2.3: Matriz de programação dinâmica para alinhamento global

Uma vez que as sequências a serem alinhadas podem ser bastante longas, é importante determinar a performance dos algoritmos utilizados em relação ao tempo de processamento e a memória necessária para realização de alinhamentos.

Para o algoritmo descrito acima pode-se perceber a matriz de programação dinâmica tem tamanho $(m+1)(n+1)$ e para obetnção do valor para cada célula é preciso fazer três operações de soma e uma operação para encontrar máximo, conforme a função de recursão da Equação (2.3). Dessa forma pode-se dizer que o algoritmo tem complexidade da ordem de $\mathcal{O}(mn)$ em tempo e espaço [9]. Considerando que as sequências s e t tem tamanhos comparáveis diz-se que a complexidade em tempo e espaço de memória do algoritmo de Needleman-Wunsh é da ordem $\mathcal{O}(n^2)$.

Na seção seguinte é apresentada uma adaptação do algoritmo NW que pode ser utilizada para encontrar alinhamentos locais ótimos entre duas sequências.

2.2.2 Algoritmo Smith-Waterman (SW)

Para aplicações em que se busca obter regiões de similaridade entre duas sequências s e t, ou o alinhamento entre subsequências de s e t, o algoritmo Smith-Waterman [10] pode ser utilizado para encontrar o alinhamento local ótimo entre essas sequências.

Esse algoritmo é adaptado de forma que sua função de recorrência tem a forma:

$$F(i,j) = \max \begin{cases} 0, \\ F(i-1,j-1) + w(x_i, y_j), \\ F(i-1,j) + w(x_i, -), \\ F(i,j-1) + w(-, y_j). \end{cases} \tag{2.4}$$

em que a alteração com a possibilidade de inserir o valor zero na matriz de programação dinâmica representa a indicação de início de novo alinhamento local.

Como os alinhamentos locais podem iniciar a partir de qualquer ponto das sequências, as bordas superior $F(0,j)$ e esquerda $F(i,0)$ da matriz são preenchidas com zeros. As demais células são calculadas de forma recursiva a partir da equação (2.4).

A Tabela 2.4 ilustra a matriz de programação dinâmica calculada para o alinhamento das sequências s e t apresentadas na Tabela 2.1, considerando as seguintes escolhas para a função *score*: *gaps* recebem valor $w(x,-) = w(-,x) = -2$, *matches* recebem valor $w(x,y) = 1$ se $x = y$ e *mismatches* recebem valor $w(x,y) = -1$ se $x \neq y$.

Para obter o alinhamento local ótimo, a etapa de *traceback* tambêm é adaptada, de forma que o início do caminho a percorrer na matriz F está na posição $i = a$ e $j = b$ tal que $F(a,b) = \max(F(x,y))$, com $x = 0, \ldots, m$, $y = 0, \ldots, m$. A partir desse ponto de máximo percorre-se o caminho ótimo até que se encontre nesse caminho uma célula tal

	-	T	C	A	G	A	C	T	A	A	T	T	C	A	T	A
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	1	0	1	0	0	1	1	0	0	0	1	0	1
T	0	1	0	0	0	0	0	1	0	0	2	1	0	0	2	0
C	0	0	2	0	0	0	1	0	0	0	0	1	2	0	0	1
T	0	1	0	1	0	0	0	2	0	0	1	1	0	1	1	0
T	0	1	0	0	0	0	0	1	1	0	1	2	0	0	2	0
G	0	0	0	0	**1**	0	0	0	0	0	0	0	1	0	0	1
A	0	0	0	1	0	**2**	0	0	1	1	0	0	0	2	0	1
C	0	0	1	0	0	0	**3**	1	0	0	0	0	1	0	1	0
T	0	1	0	0	0	0	1	**4**	2	0	1	1	0	0	1	0
A	0	0	0	1	0	1	0	2	**5**	3	1	0	0	1	0	2
A	0	0	0	1	0	1	0	0	3	**6**	4	2	0	1	0	1
G	0	0	0	0	2	0	0	0	1	4	5	3	1	0	0	0
C	0	0	1	0	0	1	1	0	0	2	3	4	4	2	0	0
A	0	0	0	2	0	1	0	0	1	1	1	2	3	5	3	1
A	0	0	0	1	1	1	0	0	1	2	0	0	1	4	4	4
T	0	1	0	0	0	0	0	1	0	0	3	1	0	2	5	3

Tabela 2.4: Matriz de programação dinâmica para alinhamento local

que $F(x, y) = 0$. Os valores destacados na Tabela 2.4 indicam o caminho para obtenção do alinhamento local ótimo α_2 exibido na Tabela 2.5.

$\alpha(s,t)$						
s^*:	G	A	C	T	A	A
t^*:	G	A	C	T	A	A

Tabela 2.5: Exemplo de Alinhamento Local

É simples notar que de forma semelhante ao algoritmo NW o algoritmo SW tambêm tem complexidade de tempo e espaço de memória da ordem $\mathcal{O}(n^2)$.

2.2.3 Algoritmo Hirschberg

O algoritmo proposto por Hirschberg [4] permite obter alinhamento global ótimo entre duas sequências em espaço linear $\mathcal{O}(n)$. Tambêm baseado no cálculo de uma matriz de programação dinâmica, esse algoritmo subdivide recursivamente o problema do alinhamento de duas sequências em problemas menores de alinhamento de subsequencias das sequências de interesse.

A subdivisão do problema de alinhamento é feita pela divisão na metade da sequência s em duas subsequências e pela divisão adequada da sequência t de forma que a concatenação dos alinhamentos ótimos das subsequências seja o alinhamento ótimo das sequências s e t, ou seja:

$$opt(s[1..i], t[1..j]) + opt(s[i+1..m], t[j+1..n]) = opt(s,t), \text{ para } j \in [1, n] \tag{2.5}$$

Para realização dessa abordagem é necessário que, em cada subdivisão do problema, encontre-se a coluna j que torne a equação (2.5) verdadeira. Para encontrar o valor de j deve-se calcular em um vetor CC o *score* do alinhamento entre um prefixo $s[1..i]$ e t, e em um vetor CC' o *score* do alinhamento entre um sufixo $s[i+1..m]$, e t. Somando os *scores* do prefixo e sufixo, por meio dos vetores CC e CC', determina-se como j a coluna que contêm o máximo da soma. Dessa forma define-se a partição do problema de alinhamento.

É importante notar que para determinação da coluna por onde passa o caminho de ótimo, as matrizes de programação dinâmica não precisam ser armazenadas em sua integralidade, uma vez que não será feito o *traceback*. Assim, basta que sejam armazenadas as últimas linhas das matrizes, nos vetores CC e CC', e dessa forma o problema tem complexidade de espaço linear $\mathcal{O}(n)$, mas mantêm complexidade de tempo quadrática $\mathcal{O}(n^2)$ e por incluir passos adicionais nas etapas intermediarias o tempo total de execução é multiplicado por uma constante próxima a 2.

Para obtenção do alinhamento ótimo pode-se continuar reduzindo o problema de forma recorrente até que ele seja trivial. A figura 2.1 exibe as subdivisões na matriz de programação dinâmica ao implementar o algoritmo de Hirschberg às sequências da Tabela 2.1.

	T	C	A	G	A	C	T	A	A	T	T	C	A	T	A
A															
T															
C															
T															
T															
G															
A															
C															
T															
A															
A															
G															
C															
A															
A															
T															

Figura 2.1: Subdivisão da Matriz de Programação Dinâmica pelo Método de Hirschberg.

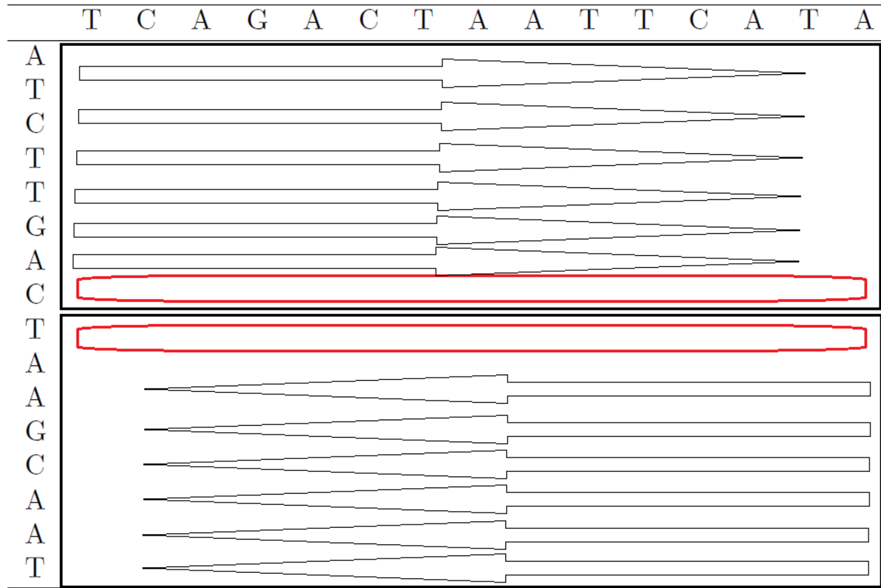

Figura 2.2: Processo para Determinação da Coluna por onde Passa o Alinhamento Ótimo.

2.2.4 Algoritmo de Gotoh

O algoritmo de Gotoh [5], em sua formulação, permite que o tamanho dos *gaps* sejam considerados no cálculo do *score* entre dois alinhamentos. O modelo *affine-gap* atribui pesos distintos para iniciar ou extender uma série de *gaps* consecutivos, de forma que um *gap* de tamanho k contribua com o *score* com $gap_k = uk + v$. Tomando $v \leq 0$ e $u \leq 0$, o custo do primeiro *gap* de uma série é dado por $u + v$, e u é o custo de cada *gap* adicional em uma série de *gaps* consecutivos.

Na etapa de programação dinâmica precisa-se de duas matrizes G e H, adicionais à matriz F, para com essas três matrizes representar as diferentes possibilidades em uma dada posição (i, j): alinhamento de $s[i]$ com $t[j]$, alinhamento de $s[i]$ com um *gap*, ou alinhamento de um *gap* com $t[j]$.

As equações de recorrência são dadas por:

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + w(x_i, y_j), \\ G(i,j), \\ H(i,j). \end{cases} \tag{2.6}$$

$$G(i,j) = \max \begin{cases} F(i-1, j) + gap_1, \\ G(i-1, j) + u. \end{cases} \tag{2.7}$$

$$H(i,j) = \max \begin{cases} F(i,j-1) + gap_1, \\ H(i,j-1) + u. \end{cases} \tag{2.8}$$

No caso de interesse em alinhamentos locais pode-se adaptar a equação de recorência para a matriz F conforme o algoritmo SW.

Apesar do aumento no número de matrizes calculadas, a ordem de complexidade de tempo e espaço do algoritmo de Gotoh mantém-se quadrática em $\mathcal{O}(mn)$. A seção seguinte apresenta uma adaptação do algoritmo de Hirschberg que permite utilizar o modelo de *affine-gap* e obter alinhamentos ótimos em espaço linear $\mathcal{O}(m+n)$.

2.2.5 Algoritmo de Myers e Miller (MM)

A proposta do algoritmo de Myers e Miller [6] é semelhante à do algoritmo de Hirshberg, subdividir o problema de alinhamento em problemas menores de alinhamentos de subsequências e concatenar os resultados para obter o alinhamento para as sequências completas. A divisão dos problemas é feita de forma análoga, dividindo a matriz de programação dinâmica na metade das linhas ou da sequência $s[1..m]$ e encontrando a coluna por onde passa o alinhamento ótimo na linha de divisão da matriz.

Uma vez que o modelo de *affine-gap* é considerado, na etapa de determinação da coluna j por onde passa o alinhamento ótimo deve-se considerar tambem os cenários e *scores* para alinhamentos que potencialmente possuem séries de *gaps*. Para isso será preciso armazenar vetores adicionais DD e DD' para obtenção em espaço linear dos *scores* dos alinhamentos do prefixo e sufixo necessários para o particionamento dos recursivos problemas de alinhamento considerados por Hirshberg [4].

Para o cálculo do *score* para o alinhamento entre um prefixo $s[1..i]$ e t são armazenados um vetor CC que representa o *score* para um alinhamento que termina com um *match* ou *mismatch*, e um vetor DD que representa o *score* para um alinhamento que termina com um *gap*. De forma analoga, no cálculo do *score* para o alinhamento entre um sufixo $s[i+1..m]$ e t são armazenados um vetor CC' que representa o *score* para um alinhamento que inicia com um *match* ou *mismatch*, e um vetor DD' que representa o *score* para um alinhamento que inicia com um *gap*.

Nos casos em que a coluna pela qual passa o alinhamento ótimo contenha um *gap* é importante notar que a penalidade pela abertura de *gap* será considerada de forma dobrada, uma vez que será contabilizada em ambos os vetores DD e DD', por isso aplica-se uma correção que define a coluna j, que determinará a partição do problema, encontrando o mínimo custo de: $K_j = \min(CC_j + CC'_j, DD_j + DD'_j - v)$.

De forma recursiva, os subproblemas obtidos pela partição encontrada podem ser subdivididos em problemas menores até que se tornem tão pequenos que tenham resolução

trivial. Assim como o algoritmo de Hirshberg, o algoritmo MM tem complexidade linear $\mathcal{O}(m+n)$ no espaço, e quadrática $\mathcal{O}(n^2)$ no tempo.

Capítulo 3

Projeto de Implementações Paralelas em CPU

Nesta seção são propostos experimentos para comparar o desempenho de diferentes implementações do algoritmo de Myers e Miller para alinhamento de sequências. As diferentes formas de implementação são descritas e o planejamento dos experimentos são apresentados.

3.1 Implementações do Algoritmo Myers e Miller

Como descrito na seção 2.2.5 o algoritmo MM tem complexidade linear no espaço, e complexidade quadrática no tempo. Com o objetivo de acelerar o tempo para alinhamento entre sequências grandes, são propostas implementações paralelas em CPU para o algoritmo MM.

A independência existente entre o cálculo do *score* para o alinhamento entre um prefixo $s[1..i]$ e t e o cálculo do *score* do alinhamento entre um sufixo $s[i+1..m]$, e t sugere uma forma simples de paralelização no algoritmo de MM. Em cada etapa do algortimo em que as sequências são subdivididas em problemas mais simples, a utilização de mais de um dos núcleos do processador deve proporcionar um ganho de velocidade no alinhamento de sequências de grandes dimensões.

Para efeito de comparação são propostas implementações paralelas e tambêm implementação sequêncial do algoritmo. Para todas as implementações propostas, a decomposição dos problemas em subsequências de tamanhos menores foi concebida por meio da organização desses subproblemas em uma fila. Em cada posição da fila está representado um problema que consiste de duas sequências a serem alinhadas. Para essas sequências é determinado um alinhamento, caso seja trivial, ou uma nova decomposição do problema

de alinhamento em dois subproblemas menores que serão posteriormente resolvidos conforme o andamento da fila.

A opção pela utilização de estrutura de fila em detrimento da proposição original de Myers e Miller, de implementação recursiva, foi considerada devido a simplicidade no controle do número de instâncias paralelas que a implementação em fila proporciona. A cada iteração pelos problemas de alinhamento contidos na fila, são definidas duas tarefas independentes, uma para calcular o *score* para o alinhamento entre um prefixo de s e t e outra para o cálculo do *score* do alinhamento entre um sufixo de s, e t. Cada uma das tarefas pode ser realizada por uma instância paralela independente.

Nesse contexto de implementação paralela do algoritmo, foram consideradas duas formas alternativas, compreendidas pelo uso de múltiplos processos ou múltiplas *threads*. Ambas as formas permitem a utilização simultânea de múltiplos processadores mas possuem diferenças de conceito e implementação que podem impactar no desempenho da aplicação. Uma vez que múltiplas *threads* contidas em um mesmo processo compartilham recursos computacionais elas são mais "leves"e por isso a criação de *threads*, em determinados sistemas, pode ser de 10 a 100 vezes mais rápida que a criação de processos [11].

A Figura 3.2 exemplifica as subdivisões consideradas nas propostas de implementação do algoritmo. No início da execução, a fila tem apenas o problema 1, que consiste na execução das tarefas independentes 1.1 e 1.2 para determinar a partição ótima do problema 1. Com os resultados dessas tarefas são criados e enfileirados os problemas 2 e 3 conforme ilustra a Figura 3.1.

Enquanto existir algum problema na fila, o algoritmo segue iterando e o próximo problema na fila deve ser resolvido de forma análoga. Sempre que um problema é tão pequeno que tem solução trivial essa solução é concatenada, e ao final da resolução de todos os problemas da fila tem-se um alinhamento ótimo entre as duas sequências de entrada do problema 1.

Instante	Fila				
0	Problema 1				
1	Problema 2	Problema 3			
2	Problema 3	Problema 4	Problema 5		

Figura 3.1: Representação do estado da fila em diferentes instantes.

O algoritmo 1 apresenta, em pseudocódigo, as etapas desenvolvidas na obtenção de alinhamento ótimo. Inicialmente na linha 1 são lidas as sequências a serem alinhadas,

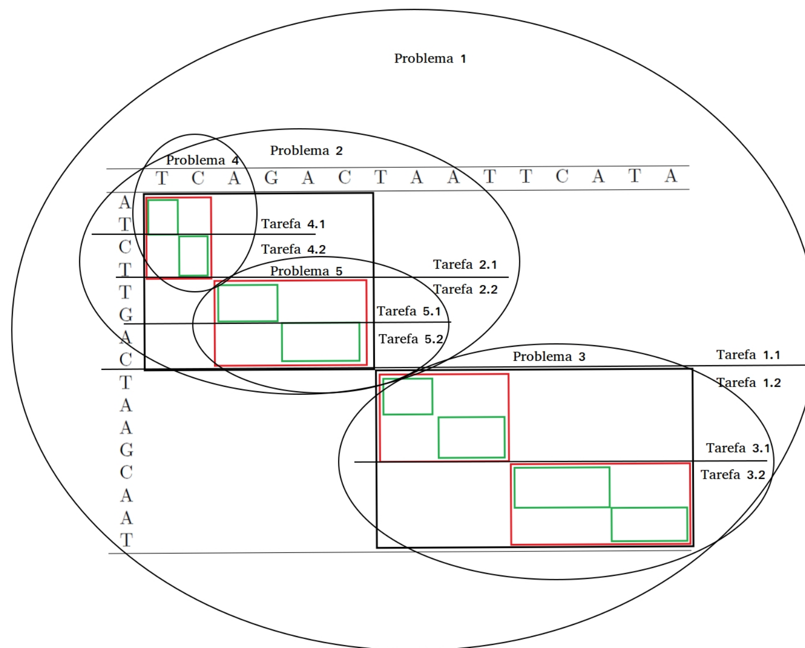

Figura 3.2: Diagrama com as subdivisões consideradas nas implementações do algoritmo de Myers e Miller.

é criada uma linha vazia (linha 2) e esse problema de alinhamento é enfileirado (linha 3). Enquanto a fila não estiver vazia o algoritmo segue na busca pelo alinhamento ótimo (linha 4). Na linha 5 avalia-se se o problema de alinhamento a ser resolvido é trivial, esse caso ocorre quando uma das sequências tem um ou menos caracteres. Na linha 6 o alinhamento trivial é adicionado à saida *output*.

Quando um alinhamento não é trivial (linha 7) é necessário determinar o particionamento ótimo em subproblemas menores de alinhamento. Para essa determinação calcula-se o score das partes inferior e superior da matriz, conforme as linhas 8 e 9. A partir dos resultados de score define-se a coluna por onde passa o alinhamento ótimo conforme a linha 10. Com a partição ótima definida nas linhas 11 e 12,os novos subproblemas são enfileirados nas linhas 13 e 14 e então o algoritmo avança na fila (linha 15) para resolver o próximo subproblema de alinhamento. Na linha 18, quando a fila já está vazia e todos os subproblemas foram resolvidos, o alinhamento ótimo entre as sequências de entrada é retornado.

As diferenças entre as três implementações propostas neste trabalho ocorrem nas etapas de realização das tarefas .1 e .2, dentro de cada problema. As linhas 8 e 9 do algoritmo 1 podem ser realizadas de forma independente e por isso, essas tarefas são realizadas alternativamente de forma sequencial ou de forma paralela utilizando dois processos ou duas

Algorithm 1 Alinhamento Global de sequências

```
 1: leia ( (Seq1, Seq2) )
 2: crieFilaVazia(q)
 3: enfileire ((q, S = Seq1, T = Seq2))
 4: while q ≠ NULL do
 5:    if alinhamento(S , T) == trivial then
 6:       output ← alinhamento(S,T)
 7:    else {encontre o particionamento ótimo fazendo}
 8:       score1 = calculaScore(parteSuperior)
 9:       score2 = calculaScore(parteInferior)
10:       col = encontraColunaOtima(score1,score2)
11:       defineSubsequenciasSuperiores(col, subSup1, subSup2)
12:       defineSubsequenciasInferiores(col, subInf1, subInf2)
13:       enfileire(q, S=subSup1, T=subSup2)
14:       enfileire(q, S=subInf1, T=subInf2)
15:    end if
16:    avanceFila(q)
17: end while
18: return output
```

threads.

Testes iniciais com a implementação paralela com dois processos indicaram que o desempenho dessa técnica poderia ser inferior ao desempenho da implementação sequencial. A estratégia do algoritmo de subdividir os problemas em partes menores até que sejam triviais depende do particionamento desses problemas utilizando algoritmo de complexidade quadrática no tempo. Nas primeiras etapas do algoritmo são decompostos problemas grandes e portanto com alto custo computacional e se beneficiam da utilização de múltiplos processadores, mas conforme o algoritmo avança os problemas que precisam ser decompostos passam a ser pequenos ou finalmente triviais. Nos casos de problemas pequenos o tempo gasto no processamento com as operações necessárias para utilização da múltiplos processos é desvantajoso em relação ao ganho pelo uso de múltiplos processos para solucionar problemas pequenos.

Para adequar a utilização do algoritmo MM com múltiplos processos, é proposta a imposição de um limite L para o tamanho da sequência dentro de uma tarefa que determine, durante a execução do algoritmo, que subproblemas com sequências de dimensão maior que L devem ser resolvidos utilizando paralelamente múltiplos processos, e subproblemas com sequências de dimensão menor que L devem ser resolvidos sequencialmente utilizando um único processo.

A determinação do valor da constante L a ser utilizada assim como a avaliação do desempenho das diferentes implementações propostas são consideradas a partir dos experimentos apresentados na seção seguinte.

3.2 Experimentos

Para determinar se existem diferenças significativas entre as médias de tempo para alinhamento de sequências utilizando as diferentes implementações consideradas foi proposto um experimento em blocos completos [12].

Para esse experimento foram considerados 5 blocos, representando 5 pares de sequências, de tamanho 10000 cada, geradas aleatóriamente com probabilidade uniforme entre os caracteres do alfabeto $\Sigma = \{A, C, T, G\}$. Cada bloco corresponde portanto a um problema de alinhamento entre duas sequências com 10000 caracteres cada.

As diferentes formas de implementação: sequencial, múltiplos-*threads* e múltiplos-processos representam os diferentes tratamentos. Cada um dos 5 problemas de alinhamento, representados pelos blocos, é resolvido pelas três diferentes implementações e dessa forma tem-se um experimento em blocos completos. A vantagem da utilização do experimento em blocos é a possibilidade de isolar as diferenças ocasionadas pelo uso de diferentes pares de sequências, uma vez que as variações ocasionadas por essas diferenças não são de interesse do estudo.

A variável resposta neste experimento é o tempo gasto, em segundos, para realizar o alinhamento entre duas sequências, e o modelo matemático que representa o experimento em questão é dado por:

$$y_{ij} = \mu + \tau_i + \beta_j + e_{ij},\ i = 1, 2, 3\ j = 1, 2, 3, 4, 5 \tag{3.1}$$

em que y_{ij} representa o tempo gasto pela i-ésima implementação para alinhar o j-ésimo par de sequências, μ é a média geral de tempo gasto pelas diferentes implementações para alinhar os diferentes pares de sequências consideradas, τ_i representa o efeito da i-ésima implementação sobre a média, β_j é efeito do j-ésimo par de sequências, e finalmente e_{ij} é o erro ou resíduo do modelo associado à observação y_{ij}. Para que os parâmetros do modelo sejam identificáveis impõe-se as restrições: $\sum_{i=1}^{3} \tau_i = 0$ e $\sum_{j=1}^{5} \beta_j = 0$.

A partir da adoção e da verificação de suposição de Normalidade e homogêneidade de variância para os resíduos do modelo, pode-se utilizar a técnica de Análise de Variância para determinar se existem diferenças significatívas entre as médias de tempo das diferentes implementações [12]. Caso tais suposições não sejam válidas é possível a utilização de teste não paramétrico de Friedman [13] para testar a existência de diferenças significativas entre as diferentes implementações.

Para ambos os casos paramétrico e não-paramétrico, as hipóteses em teste são definidas como:

$$\begin{cases} H_0: & \text{Não existem diferenças entre as médias de tempo das diferentes implementações.} \\ H_A: & \text{Existem diferenças entre as médias de tempo das diferentes implementações.} \end{cases}$$

ou alternativamente utilizando a representação do modelo na equação 3.1

$$\begin{cases} H_0: \tau_1 = \tau_2 = \tau_3 = 0 \\ H_A: \exists \tau_i \neq 0 \quad i = 1, 2, 3 \end{cases}$$

Caso a hipótese de igualdade seja rejeitada testes adicionais como TukeyHSD [12] ou Mann-Whitney [13] podem ser realizados para determinar específicamente quais tratamentos ou implementações são diferentes entre si.

Alem do experimento em blocos descrito, propõe-se tambêm explorar o desempenho das diferentes implementações em relação a diferentes tamanhos das sequências a serem alinhadas. Para avaliar o efeito da dimensão das sequências sobre as possíveis diferenças nos tempos entre as distintas implementações, são propostos pares de sequências com tamanhos: 5000, 10000, 15000 e 20000.

Para cada um dos diferentes pares deve-se registrar o tempo decorrido para o alinhamento considerando cada uma das três propostas de implementação. A partir dos resultados pode-se avaliar se houve uma aceleração ou *speedup* entre tempos para alinhamento considerando as implementações paralelas em relação à implementação sequêncial e o efeito da dimensão das sequências a serem alinhadas.

Uma vez que a implementação proposta utilizando múltiplos processos depende da atribuição de um valor limite L, um experimento é proposto para determinar um valor adequado para esse limite. Para diferentes valores desse limite são calculados os tempos para alinhamento de um par de sequências com tamanho 10000.

A seção seguinte apresenta o resultado dos experimentos realizados.

Capítulo 4

Resultados Experimentais

Neste capítulo são apresentados os resultados obtidos nos experimentos descritos na Seção 3.2 e realizados em sistema com as seguintes configurações: Notebook HP com processador Intel Core i7-4510U (4 cores) 2.00GHz, 8 GB RAM, com sistema operacional Ubuntu 16.04 LTS. As implementações foram desenvolvidas em linguagem C e compilador gcc versão 5.4.0. Os gráficos e análises nesta seção foram realizados utilizando o ambiente e linguagem R de programação estatística [14].

4.1 Determinação do Limite L para Utilização na Implementação Multi-processos

Para determinar um valor L adequado para a implementação proposta com múltiplos processos foram testados os valores: (1, 5, 10, 25, 50, 75, 100, 125, 150, 175, 200, 225, 250). Para cada caso foram medidos os tempos para 5 repetições do alinhamento de um par de sequências de tamanho 10000 geradas aleatóriamente.

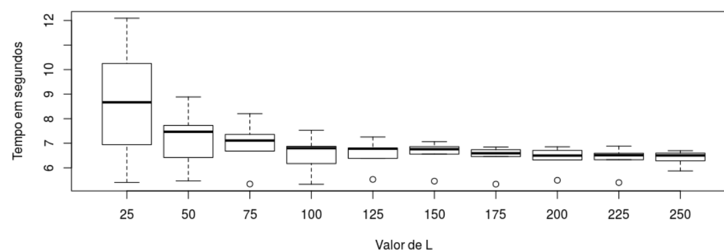

Figura 4.1: Distribuições dos tempos para alinhamento de sequências de tamanho 10000 para diferentes valores de L

A Figura 4.1 apresenta os resultados que indicam que para valores a partir de $L = 100$ os tempos se tornam estáveis. Os resultados para valores de L inferiores a 25 não estão apresentados no gráfico pois tiveram desempenho muito inferior aos demais casos, tendo tempos de execução muito acima de 50 segundos e desta forma prejudicando a visibilidade no gráfico dos efeitos de L na região de interesse. Assim, nos experimentos com a implementação proposta com múltiplos processos será considerado o valor de $L = 100$.

4.2 Avaliação de Diferença entre as Implementações

O experimento em blocos completos tem seus resultados apresentados na Tabela 4.1. Cada par de sequências (bloco) foi alinhado de forma repetida dez vezes, com cada uma das implementações, para diminuir o efeito da variabilidade no tempo de execução observado entre repetições de um mesmo procedimento.

Para fins de análise do experimento são utilizadas as médias obtidas entre as repetições, conforme apresentadas na Tabela 4.1, juntamente com os desvios-padrões dos tempos observados entre as repetições.

Tabela 4.1: Médias e (Desvios-padrões) dos tempos, em segundos, para alinhamento de sequências de tamanho 10000

Implementação	Bloco 1	Bloco 2	Bloco 3	Bloco 4	Bloco 5
Sequencial	8.1 (0.06)	8.1 (0.06)	8.07 (0.06)	8.08 (0.05)	8.1 (0.05)
MultiThread	5.67 (0.05)	5.65 (0.02)	5.66 (0.07)	5.63 (0.02)	5.66 (0.03)
MultiProcesso	5.38 (0.07)	5.37 (0.03)	5.39 (0.09)	5.43 (0.13)	5.38 (0.05)

A Figura 4.2 ilustra as distribuições dos tempos para alinhamentos de sequências de tamanho 10000 para as três implementações utilizadas. Pode-se perceber no gráfico a diferença na performance das diferentes propostas. Diferença considerada significativa pela análise de variância e pelo teste de Friedman ($pvalor < 0,01$) que indica para rejeição da hipótese de igualdade entre as médias das implementações propostas.

Os resultados de testes adicionais de Tukey e Mann-Whitney com $\alpha = 0,05$ indicaram que a implementação com múltiplos-processos é significativamente mais rápida que a implementação com múltiplos-*threads*, que por sua vez é significativamente mais rápida que a implementação sequencial.

Vale ressaltar que as suposições adotadas para a análise de variância e testes de comparações múltiplas paramétricas foram confirmadas e os resultados das técnicas paramétricas e não paramétricas foram conconrdantes.

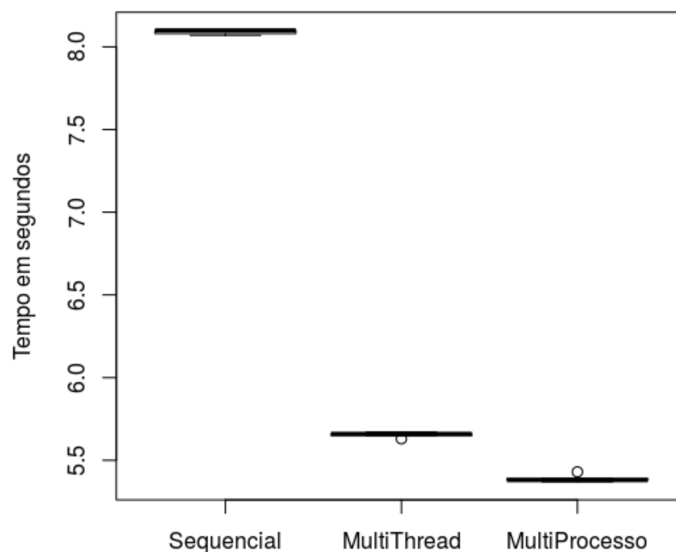

Figura 4.2: Distribuições dos tempos para alinhamento de sequências de tamanho 10000

4.3 Avaliação de Ganho de Desempenho pelas Implementações Paralelas

A Figura 4.3 ilustra o ganho de performance ou *speedup* das implementações paralelas em relação à implementação sequencial. Pode-se notar que para todos os tamanhos de sequências considerados houve ganho de performance pela utilização de programação paralela. Para sequências com dimensão a partir de 10000 as implementações paralelas são aproximadamente 1.5 vezes mais rápidas do que a implementação sequencial.

É interessante observar que apesar das implementações paralelas permitirem uso simultâneo de 2 processadores o tempo de execução não é reduzido pela metade devido ao *overhead* relativo ao uso das ferramentas para processamento paralelo. Os resultados deste experimento também indicam melhor desempenho da técnica com múltiplos processos em relação à implementação com múltiplas *threads*.

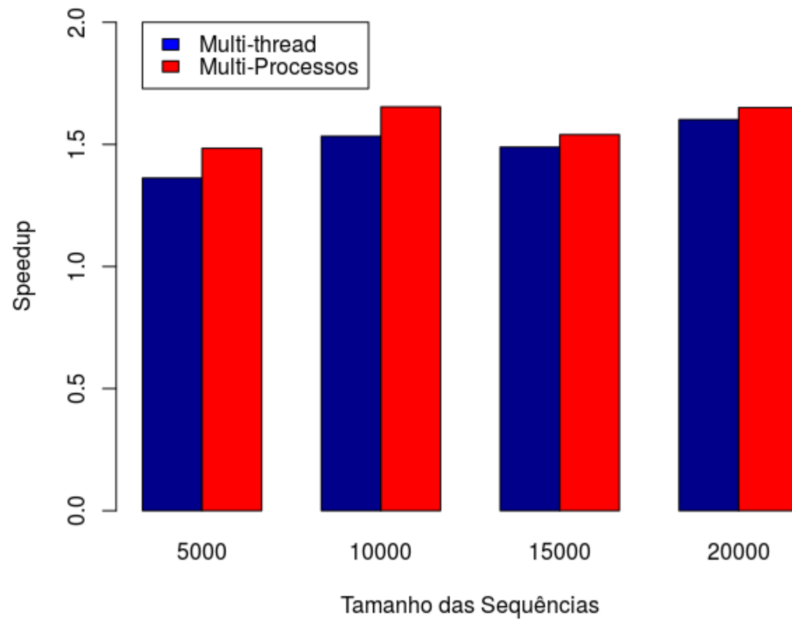

Figura 4.3: Avaliação do ganho de performance *speedup* pelo uso de paralelização

Tabela 4.2: Médias e (Desvios-padrões) dos tempos, em segundos, para alinhamento de sequências de tamanhos variáveis

results	5000	10000	15000	20000
Sequencial	2.33 (0.05)	8.88 (0.31)	18.05 (0.1)	33.41 (1.29)
MultiThread	1.71 (0.01)	5.79 (0.18)	12.12 (0.63)	20.86 (0.62)
MultiProcesso	1.57 (0.03)	5.37 (0.03)	11.72 (0.43)	20.24 (0.65)

Capítulo 5

Conclusões e Trabalhos Futuros

5.1 Conclusões

Neste trabalho foram considerados problemas de alinhamento global de sequências biológicas. Para essa categoria de problemas foram apresentadas propostas de implementação paralelas, em cpu, para o algoritmo de Myers-Miller com o objetivo de acelerar os tempos de alinhamento entre sequências com 5000 a 20000 caracteres. As propostas apresentadas foram comparadas com uma implementação sequencial do mesmo algoritmo.

As diferentes implementações paralelas consideram utilização de múltiplos processos ou múltiplas *threads*. A utilização de múltiplos processos pode ser desvantajosa devido ao *overhead* relativo à criação de novos processos e por isso foi proposta a adoção de um limite L durante a implementação com múltiplos processos. Esse limite L determina pelo tamanho de um subproblema de alinhamento se esse problema deve ser resolvido de forma paralela com dois processos ou de forma sequencial.

Um pequeno experimento foi conduzido para determinação de um valor adequado para L, indicando que os valores entre 100 e 250 tiveram tempos semelhantes e estáveis para realização dos alinhamentos. Nos demais experimentos considerados no trabalho o valor utilizado foi $L = 100$.

A característica das *threads* serem mais leves que os processos foi confirmada pelo fato que o *overhead* observado na implementação com múltiplos processos não se repetiu na implementação com múltiplas *threads* e dessa forma todos os subproblemas foram resolvidos utilizando duas *threads*, independente de quão pequeno fosse o subproblema.

As médias de tempo para alinhamento obtidas pelas as implementações paralelas foram consideradas significativamente menores que a média de tempo para alinhamento obtida pela implementação sequencial, conforme esperado. Os testes estatísticos indicaram que as implementações paralelas são significativamente mais rápidas do que a implementação

sequencial, e a implementação com múltiplos processos foi significativamente mais rápida do que a implementação com múltiplas *threads*.

Para sequências com tamanho a partir de 5000 as implementações paralelas são 1.5 vezes mais rápidas que a implementação sequencial. Ainda que tenham sido utilizados 2 processadores com as implementações paralelas as propostas não foram 2 vezes mais rápidas que a implementação sequencial devido aos custos associados à utilização das ferramentas para execução em paralelo.

5.2 Trabalhos futuros

A partir da execução desse trabalho foram destacadas algumas possibilidades de extensão ou adaptação das idéias aqui apresentadas em trabalhos futuros.

- Implementação com múltiplas tarefas paralelas (Não limitada a 2): sugerimos que sejam criadas tantas *threads* quantos cores existirem na máquina, de maneira a explorar mais efetivamente o paralelismo. Para tanto, deve ser proposta uma estratégia de alocação balanceada de trabalho entre as *threads*.

- Implementação em hardware reconfigurável: ao estudar o estado-da-arte, notamos que não existe implementação do algoritmo Myers-Miller em FPGA (Field Programmable Gate Arrays). Sugerimos, portanto, sua implementação em tal plataforma com particionamentos do circuito a cada recursão, de maneira a melhor aproveitar o paralelismo e a área do FPGA.

Referências

[1] Luscombe, Nicholas, Dov Greenbaum e M Gerstein: *What is bioinformatics? a proposed definition and overview of the field.* páginas 83–100, janeiro 2001. 1

[2] Mount, D.W.: *Bioinformatics: Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, 2001. 1, 4

[3] Needleman, Saul B. e Christian D. Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology, 48(3):443 – 453, 1970. 2

[4] Hirschberg, D. S.: *A linear space algorithm for computing maximal common subsequences.* Commun. ACM, 18(6):341–343, junho 1975, ISSN 0001-0782. 2, 9, 12

[5] Gotoh, O: *An improved algorithm for matching biological sequences.* Journal of molecular biology, 162 3:705–8, 1982. 2, 11

[6] Myers, Eugene W. e Webb Miller: *Optimal alignments in linear space.* Computer applications in the biosciences : CABIOS, 4 1:11–7, 1988. 2, 12

[7] Sandes, Edans Flavius De Oliveira, Azzedine Boukerche e Alba Cristina Magalhaes Alves De Melo: *Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification.* ACM Comput. Surv., 48(4):63:1–63:36, março 2016, ISSN 0360-0300. `http://doi.acm.org/10.1145/2893488`. 2, 3

[8] González-Pérez, Beatriz, Victoria López e Juan Sampedro: *Programming Global and Local Sequence Alignment by Using R*, páginas 341–352. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. 4

[9] Durbin, R.: *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Biological Sequence Analysis: Probabalistic Models of Proteins and Nucleic Acids. Cambridge University Press, 1998. 5, 8

[10] Smith, T.F. e M.S. Waterman: *Identification of common molecular subsequences.* Journal of Molecular Biology, 147(1):195 – 197, 1981. 8

[11] Andrew S. Tanenbaum, Herbert Bos: *Modern operating systems.* Pearson, 4ed. edição, 2014. 15

[12] Montgomery, Douglas C.: *Design and Analysis of Experiments.* Wiley, 8ª edição, 2012. 18, 19

[13] Myles Hollander, Douglas A. Wolfe, Eric Chicken: *Nonparametric Statistical Methods*. John Wiley Sons, 3rd edição, 2014. 18, 19

[14] R Core Team: *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017. `https://www.R-project.org/`. 20