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

Comparação Paralela Exata de Seqüências Biológicas Longas com Uso Limitado de Memória

Rodolfo Bezerra Batista

Dissertação apresentada como requisito parcial
para obtenção do grau de Mestre em Informática

Orientadora
Profª. Drª. Alba Cristina Magalhães Alves de Melo

Brasília
2006

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Li Weigang

Banca examinadora composta por:

Profª. Drª. Alba Cristina Magalhães Alves de Melo (Orientadora) – CIC/UnB
Prof. Dr. Wellington Santos Martins – CMP/UCG
Prof. Dr. Ricardo Pezzuol Jacobi – CIC/UnB

CIP – Catalogação Internacional na Publicação

Rodolfo Bezerra Batista.

Comparação Paralela Exata de Seqüências Biológicas Longas com Uso Limitado de Memória/ Rodolfo Bezerra Batista. Brasília : UnB, 2006. 105 p. : il. ; 29,5 cm.

Tese (Mestre) – Universidade de Brasília, Brasília, 2006.

1. bioinformática, 2. alinhamento de seqüências, 3. programação paralela

CDU 004

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

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

Comparação Paralela Exata de Seqüências Biológicas Longas com Uso Limitado de Memória

Rodolfo Bezerra Batista

Monografia apresentada como requisito parcial
para obtenção do grau de Mestre em Informática

Profª. Drª. Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Wellington Santos Martins
CMP/UCG

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Li Weigang
Coordenador do Mestrado em Informática

Brasília, 20 de março de 2006

Dedicatória

Dedico este trabalho à minha esposa Angélica, minha fonte de inspiração, cuja presença alegre e sorriso sincero são um incentivo inigualável a todas as horas.

Agradecimentos

Agradeço a toda minha família, em especial a meu pai e à minha mãe, por terem me dado todo o suporte necessário sem o qual este trabalho não seria possível.

Agradeço a Alba Cristina Melo, professora da Universidade de Brasília, pela sua orientação ímpar, tendo muitas vezes confiado na minha capacidade mais do que eu mesmo.

Agradeço a Wellington Santos Martins, professor da Universidade Católica de Goiás, e a Ricardo Pezzuol Jacobi, professor da Universidade de Brasília, por terem aceitado participar da banca de defesa desta dissertação e pelas modificações sugeridas a este documento.

Agradeço a Gerson Pfitscher, professor da Universidade de Brasília, pela sua valiosa ajuda com o cluster do Instituto de Exatas da Universidade de Brasília, equipamento no qual foram realizados os testes da implementação.

Agradeço a Antonio Bento Miranda por ter me ajudado a manter o ritmo neste trabalho, tendo por várias vezes me lembrado de dedicar o tempo de forma mais equilibrada entre obrigações profissionais e acadêmicas.

Resumo

O alinhamento de seqüências biológicas é um método muito importante usado pela biologia computacional para relacionar organismos e compreender os processos evolutivos envolvidos entre eles. O algoritmo de Smith-Waterman, método exato para obtenção de alinhamentos locais ótimos entre seqüências de DNA (ácido desoxirribonucleico), possui complexidade $O(n^2)$ tanto de espaço quanto de tempo. Esta complexidade é um obstáculo à comparação de seqüências muito longas. O BLAST é uma ferramenta capaz de produzir alinhamentos locais em curto espaço de tempo e baixo custo de memória. No entanto, a sensibilidade dos resultados produzidos é baixa em comparação aos métodos exatos, devido às heurísticas utilizadas no BLAST.

A programação paralela é utilizada para lidar com problemas computacionais que demandam muito tempo de processamento. *Clusters* de computadores provêm alto poder computacional a baixo custo. Entretanto, para se ter benefícios com o uso de *clusters*, os problemas precisam ser adaptados antes de serem resolvidos sobre tal plataforma computacional.

A presente dissertação propõe uma estratégia paralela exata para a comparação de seqüências longas de DNA em um espaço limitado de memória. A estratégia proposta foi implementada em um *cluster* de estações de trabalho, atingindo *speedups* muito bons para seqüências maiores que 50Kbp e sendo capaz de produzir alinhamentos locais ótimos para seqüências de mais de 3 milhões de pares de bases.

Palavras-chave: bioinformática, alinhamento de seqüências, programação paralela

Abstract

Biological sequence alignment is a very important method used by computational biology to relate organisms and undestand the evolutionary processes involved between them. The Smith-Waterman algorithm, an exact method used to obtain optimal local alignments between DNA (deoxyribonucleic acid) sequences, has $O(n^2)$ space and time complexity. This complexity is an obstacle to the comparison of very long sequences. BLAST is a tool capable of producing local alignments in short time at a low memory cost. However, the results produced have a low sensibility when compared to exact methods, due to the heuristics used in BLAST.

Parallel programming is used to deal with high processeing time demanding computational problems. Clusters of computers provide high computational power at low cost. However, in order to have benefits with the use of clusters, the problems must be adapted before being solved on such computational platform.

The present dissertation proposes an exact parallel strategy to the comparison of long DNA sequences in limited memory space. The proposed strategy was implemented in a cluster of workstations, reaching very good speedups for sequences longer than 50Kbp and being able to produce optimal local alignments for sequences with over 3 million base pairs.

Keywords: bioinformatics, sequence alignment, parallel programming

Sumário

Lista de Figuras **10**

Lista de Tabelas **12**

Capítulo 1 Introdução **13**
1.1 Motivação 13
1.2 Contribuição 14
1.3 Organização 15

Capítulo 2 Comparação de seqüências biológicas **16**
2.1 Conceitos básicos de Genética 16
2.1.1 Células 16
2.1.2 Proteínas 17
2.1.3 DNA 20
2.1.4 RNA 22
2.1.5 Síntese Protéica 22
2.2 Comparação e alinhamento entre seqüências genéticas 25
2.3 Alinhamento global 26
2.4 Alinhamento local 29
2.5 *Gaps* 30
2.5.1 Gotoh 31
2.6 Aprimoramentos sobre os algoritmos de comparação de seqüências 33
2.6.1 Reduzindo o custo de memória 33
2.6.2 Reduzindo o custo de tempo 36

Capítulo 3 Computação em *clusters* **38**
3.1 Visão Geral 38
3.1.1 *Clusters* 38
3.1.2 Características dos *clusters* 38
3.1.3 Taxonomia 41
3.2 Programação Paralela 45
3.2.1 Modelos de programação 45
3.2.2 Projeto de algoritmos paralelos 46

Capítulo 4 Soluções paralelas e distribuídas **48**
4.1 Prefixo paralelo 50
4.2 Modelo EARTH 52

4.3 PSW-DC (*Divide and Conquer*) 54
4.4 Block-Cyclic *wavefront* 56
4.5 FastLSA 58
4.6 GenomeDSM 60
4.7 Estratégias paralelas em CoW (*Cluster of Workstations*) 62
4.8 Quadro comparativo 65

Capítulo 5 Projeto da estratégia *z*Align **67**
5.1 Estratégia proposta 67
5.1.1 1ª Fase: Distribuição 67
5.1.2 2ª Fase: Programação dinâmica 68
5.1.3 3ª Fase: Agregação 73
5.1.4 4ª Fase: Alinhamento 73

Capítulo 6 Resultados experimentais **80**
6.1 Ambiente de implementação e testes 80
6.2 Seqüências analisadas 81
6.3 Medidas de desempenho paralelo 83
6.3.1 *Speedups* relativos 83
6.3.2 Tempos de comunicação MPI 85
6.4 Reprocessamento na fase de alinhamento 89
6.5 Comparações com o BLAST 90
6.6 Análise dos resultados obtidos 97

Capítulo 7 Conclusão e Trabalhos futuros **99**
7.1 Conclusão 99
7.2 Trabalhos futuros 99

Referências **101**

Lista de Figuras

2.1 Estrutura básica de um aminoácido 17
2.2 Ligação Peptídica 19
2.3 Planos rotacionais ϕ e ψ 19
2.4 Molécula de Desoxirribose 20
2.5 Bases Nitrogenadas 20
2.6 Fita dupla de DNA 21
2.7 Molécula de Ribose 22
2.8 Base nitrogenada Uracila 22
2.9 Fita de mRNA 23
2.10 Exemplo de seqüência de DNA e extremidades 5' e 3' 23
2.11 Exemplo de obtenção de um valor de similaridade 26
2.12 1º passo do algoritmo de Needleman-Wunsch 28
2.13 2º passo do algoritmo de Needleman-Wunsch 28
2.14 Matriz de similaridades $S_{i,j}$ do algoritmo de Needleman-Wunsch . 29
2.15 Matriz de similaridades $S_{i,j}$ do algoritmo de Smith-Waterman . . 30
2.16 Relações de dependência entre D, P e Q 32
2.17 Processamento do algoritmo de Hirschberg 35
2.18 Processamento do algoritmo de Fickett 37

3.1 Taxonomia de sistemas paralelos e distribuídos 43
3.2 Rede de interconexão em barramento 44
3.3 Rede de interconexão comutada 44

4.1 Processamento em onda 49
4.2 Processamento da matriz de similaridades utilizando EARTH . . . 53
4.3 Relações possíveis entre dois alinhamentos no PSW-DC 55
4.4 Divisão em colunas para $S_{i,j}$ 57
4.5 Método *block-cyclic* para processamento em onda 57
4.6 Fases do algoritmo FastLSA 59
4.7 Distribuição do trabalho no GenomeDSM. 62
4.8 Distribuição de trabalho em blocos e bandas de passagem 63
4.9 Bandas de passagem. 64
4.10 Algoritmo para redução da complexidade de espaço. 65

5.1 Divisão da matriz de similaridades 68
5.2 Linhas e colunas de passagem 69
5.3 Ordem de processamento dos blocos 71

5.4 Alinhamento, diagonais e divergências 72
5.5 Cálculo da divergência 73
5.6 Alinhamento entre S e T e suas inversas 75
5.7 Múltiplos alinhamentos ótimos 76
5.8 Alocação de memória 76
5.9 Linhas-cache 78
5.10 Visão geral do *z*Align 79

6.1 *Speedups* relativos do *z*Align 84
6.2 *Breakdown* da execução para seqüências de 1 Kbp. 85
6.3 *Breakdown* da execução para seqüências de 10 Kbp. 86
6.4 *Breakdown* da execução para seqüências de 50 Kbp. 86
6.5 *Breakdown* da execução para seqüências de 150 Kbp. 87
6.6 *Breakdown* da execução para seqüências de 500 Kbp. 87
6.7 *Breakdown* da execução para seqüências de 1 Mbp. 88
6.8 *Breakdown* da execução para seqüências de 3 Mbp. 89
6.9 Exemplos de saída do *z*Align e do BLAST 91
6.10 Alinhamentos obtidos para seqüências de 1 Kbp 92
6.11 Alinhamentos obtidos para seqüências de 10 Kbp 92
6.12 Detalhe dos alinhamentos obtidos para seqüências de 10 Kbp . . . 93
6.13 Alinhamentos obtidos para seqüências de 50 Kbp 93
6.14 Detalhe dos alinhamentos obtidos para seqüências de 50 Kbp . . . 94
6.15 Alinhamentos obtidos para seqüências de 150 Kbp 94
6.16 Alinhamentos obtidos para seqüências de 500 Kbp 95
6.17 Alinhamentos obtidos para seqüências de 1 Mbp 95
6.18 Detalhe dos alinhamentos obtidos para seqüências de 1 Mbp . . . 96
6.19 Alinhamentos obtidos para seqüências de 3 Mbp 96
6.20 Detalhe dos alinhamentos obtidos para seqüências de 3 Mbp . . . 97

Lista de Tabelas

2.1 Aminoácidos encontrados na natureza 18
2.2 Código Genético 24
2.3 Diferenças entre o DNA e o RNA 25

4.1 Comprimentos de algumas seqüências de DNA 48
4.2 Quadro comparativo das soluções paralelas 66

6.1 Subdivisões horizontais e verticais para os testes realizados. 80
6.2 Seqüências analisadas 82
6.3 Tempos de execução 84
6.4 Speedups relativos do *z*Align 84
6.5 Divergência máxima e reprocessamento 90
6.6 Quantidades e comprimentos de alinhamentos encontrados 90

Capítulo 1

Introdução

1.1 Motivação

As últimas décadas foram marcadas por enormes avanços no campo da Biologia Molecular, área da Biologia que estuda os mecanismos bioquímicos relacionados aos ácidos nucleicos e proteínas[47].

O Projeto Genoma Humano é um dos exemplos mais famosos de estudo nesta área. Tinha como objetivo seqüenciar o código genético humano, tendo este atingido a sua meta antes mesmo da data prevista. Isto se deveu, entre outros fatores, graças ao auxílio da Bioinformática.

A Bioinformática é uma área interdisciplinar que alia o poder computacional aos estudos da Biologia. O fruto desta união de forças é o surgimento de ferramentas capazes de seqüenciar, comparar e mapear o código genético dos seres vivos. Esta união de forças tem produzido ótimos resultados até então e ainda pode-se esperar muito mais dela.

É fácil justificar esta expectativa se o Projeto Genoma Humano for tomado como exemplo. Este produziu ,por meio de métodos de seqüenciamento genético, uma quantidade muito grande de informação. A partir desta informação é que os cientistas farão comparações e análises a fim de compreender com maior profundidade ainda vários dos mecanismos envolvidos durante ciclo de vida celular.

Tal compreensão será fruto de pesquisas realizadas em várias áreas conjugadas à Biologia Molecular. Dentre estas, pode-se ressaltar a Proteômica, área que estuda o conjunto de proteínas expressadas pelos organismos e os mecanismos envolvidos neste processo. Um dos mecanismos utilizados para entender melhor a expressão de certas proteínas é a análise do código genético que dá origem a elas. Uma das análises realizadas com tal finalidade é a comparação de seqüências genéticas, onde seqüências recém-seqüenciadas são comparadas com outras seqüências conhecidas a fim de se compreender quais as semelhanças e diferenças entre elas.

O BLAST[2] e o FASTA[43] são exemplos de algoritmos utilizados para realizar rapidamente estas comparações. Acontece que, para produzir seus resultados em pouco tempo, estes algoritmos fazem uso de heurísticas que prejudicam a qualidade dos resultados obtidos. Quando se deseja resultados precisos ao comparar seqüências genéticas, costuma-se optar pelo algoritmo de Smith-Waterman[49] ou

por variantes do mesmo, cuja desvantagem é o alto custo de tempo e memória. Este custo é tão grande que uma comparação entre duas seqüências completas de cromossomos humanos utilizando o algoritmo de Smith-Waterman chega a ser praticamente inviável.

A fim de possibilitar a obtenção de resultados em um tempo aceitável com o algoritmo de Smith-Waterman, várias abordagens têm sido propostas utilizando processamento paralelo. As soluções vão desde implementações em *software*, utilizando Memória Compartilhada Distribuída[36] e Passagem de Mensagens[46], a arquiteturas de *hardware*[23]. Mesmo para o caso do BLAST, que é capaz de produzir resultados rapidamente, têm sido estudadas soluções no sentido de acelerar as comparações de seqüências genéticas[30].

O custo cada vez menor dos computadores pessoais acabou por dar origem a uma classe de supercomputadores denominada *clusters*. Os *clusters* têm permitido que cada vez mais de instituições sejam capazes de construir supercomputadores a partir de equipamentos de baixo custo. Não é apenas a relação custo/benefício dos *clusters* que os torna singulares, mas também a forma de programá-los. Utilizar um *cluster* para acelerar a obtenção de resultados para certos problemas requer muitas vezes que estes sejam adaptados a tal arquitetura. Esta adaptação, muitas vezes, já é um desafio por si só. Os *clusters* de computadores têm sido muito utilizados recentemente a fim de se produzir rapidamente informações úteis aos projetos em Biologia Molecular e Proteômica.

A presente dissertação de mestrado aborda os conceitos de alinhamento de seqüências genéticas e de programação paralela a fim de produzir uma ferramenta capaz de realizar a comparação exata de seqüências biológicas com 3 milhões de pares de bases em um *cluster* com memória limitada. Tal comparação, se realizada com o algoritmo original de Smith-Waterman, necessitaria de, no mínimo, $8,18$ TB de memória.

1.2 Contribuição

A presente dissertação propõe uma estratégia capaz produzir alinhamentos ótimos para seqüências biológicas de três milhões de pares de bases utilizando um método de comparação exata em um espaço limitado de memória e processamento paralelo, para diminuir o tempo necessário para tal.

Os resultados obtidos com seqüências de DNA reais mostram que a estratégia paralela proposta é capaz de obter *speedups* superlineares para seqüências de 150.000 e 500.000 pares de bases, com 16 processadores. Além disso, a estratégia proposta foi capaz de obter alinhamentos ótimos para genomas com mais de 1 milhão de pares de bases.

A comparação entre os genomas de *Chlamydia trachomatis* A/HAR-13 [8] e de *Chlamydia muridarum* Nigg[51], ambos com aproximadamente 1 milhão de pares de bases, foi obtida em 1 hora e 43 minutos em um *cluster* de 16 processadores AMD AthlonMP 1900+. A comparação entre os genomas de *Corynebacterium efficiens* YS-314 e de *Corynebacterium glutamicum* ATCC 13032[29], ambos com aproximadamente 3 mihões de pares de bases, levou 13 horas e 31 minutos sobre a mesma plataforma.

1.3 Organização

O restante desta dissertação encontra-se organizado da seguinte forma. O Capítulo 2 apresenta conceitos fundamentais associados à Biologia Molecular e Genética.

O Capítulo 3 apresenta conceitos fundamentais acerca dos *clusters* e da programação paralela e distribuída.

O Capítulo 4 apresenta o estado da arte no que se refere ao alinhamento de seqüências genéticas em paralelo.

No Capítulo 5 é proposta uma estratégia exata capaz de alinhar seqüências grandes em paralelo usando uma quantidade limitada de memória.

O Capítulo 6 apresenta os resultados experimentais obtidos a partir da implementação da estratégia proposta.

No Capítulo 7 são apresentadas as conclusões obtidas nesta dissertação e são apresentadas sugestões de trabalhos futuros.

Capítulo 2

Comparação de seqüências biológicas

2.1 Conceitos básicos de Genética

2.1.1 Células

Podemos observar nos seres vivos uma série de fenômenos que caracterizam a presença de vida. Dentre tais fenômenos podemos citar movimento, calor, reprodução e crescimento. Tais fenômenos são macroscópicos, podendo ser observados e percebidos naturalmente. A origem destes fenômenos, no entanto, dá-se em nível microscópico. Neste nível microscópico encontram-se as células.

Dentro das células ocorre uma enorme quantidade de reações químicas que são responsáveis, por exemplo, pela produção de energia necessária para o funcionamento da célula. A estas reações químicas intracelulares dá-se o nome de metabolismo celular.

As células possuem estruturas comuns bem definidas. Possuem uma membrana plasmática que delimita o interior e o exterior da mesma. No interior das células, encontra-se o citoplasma. O citoplasma é composto por uma série de organelas especializadas, onde podemos citar o retículo endoplasmático, ribossomos, centríolos e complexo de Golgi. Tais organelas são constituídas por estruturas moleculares complexas obtidas pela combinação de elementos formadores básicos.

Os elementos que constituem as células são basicamente: água, proteínas, lipídeos, açúcares e sais minerais. O elemento mais abundante é a água, contribuindo com mais de 60% do volume celular. Logo em seguida, as proteínas ocupam o segundo lugar em volume na célula e são encontradas em praticamente todas as estruturas que a compõem.

A síntese protéica é o processo de formação de proteínas. Tal processo é regulado pela informação genética da célula. Em seres procariontes, esta informação genética está dispersa no interior da célula. Já em seres eucariontes, o conteúdo genético encontra-se no núcleo da célula, em uma estrutura denominada cromatina.

A importância de se estudar as proteínas e seu mecanismo de formação deve-

se ao fato de estas participarem de praticamente todas as reações químicas do metabolismo celular.

2.1.2 Proteínas

Proteínas são os elementos primordiais que compõem os seres vivos. Realizam várias tarefas distintas, dentre elas:

- Constituem uma grande parte da parede celular, sendo assim os componentes estruturais dos seres vivos;
- Funcionam como enzimas, assumindo a função de catalisadores que aceleram reações químicas que de outra forma ocorreriam a uma velocidade incapaz de sustentar a vida;
- Atuam como sinalizadores, indicando quando uma determinada seqüência genética deve se expressar ou não;
- Servem como sensores, reagindo a estímulos e detectando a presença de intrusos no organismo.
- Constituem moléculas como a hemoglobina, capaz de transportar moléculas de O_2 ou de CO_2 indispensáveis no processo de respiração celular.
- Participam da formação de antígenos capazes de defender a célula de ataques de vírus ou bactérias.

As proteínas são essencialmente uma cadeia de aminoácidos. Um aminoácido é representado pela estrutura química da Figura 2.1. A estrutura do aminoácido possui um átomo de carbono central, chamado Cα , ligado a uma amina (NH3), uma carboxila (COOH) e uma cadeia lateral, representada por R. Os aminoácidos são diferenciados pelas suas cadeias laterais (R).

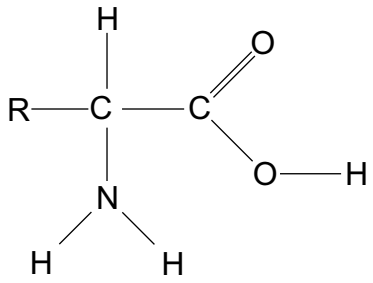


Figura 2.1: Estrutura básica de um aminoácido

Existem na natureza 20 aminoácidos diferentes. Na Tabela 2.1 vemos os aminoácidos com suas respectivas representações simbólicas. Cada aminoácido possui características peculiares devido à sua cadeia lateral.

Para compor uma proteína, vários aminoácidos são ligados através de uma ligação peptídica. A Figura 2.2 mostra uma ligação peptídica. Neste tipo de

Nome	**Símbolo (3 letras)**	**Símbolo (1 letra)**
Alanina	Ala	A
Cisteína	Cys	C
Ácido aspártico	Asp	D
Ácido glutâmico	Glu	E
Fenilalanina	Phe	F
Glicina	Gli	G
Histidina	His	H
Isoleucina	Ile	I
Lisina	Lys	K
Leucina	Leu	L
Metionina	Met	M
Asparagina	Asn	N
Prolina	Pro	P
Glutamina	Gln	Q
Arginina	Arg	R
Serina	Ser	S
Treonina	Thr	T
Valina	Val	V
Triptofano	Trp	W
Tirosina	Tyr	Y

Tabela 2.1: Aminoácidos encontrados na natureza

ligação, uma molélula de hidróxido (OH^-) desliga-se da carboxila de um aminoácido enquanto que um átomo de hidrogênio (H^+) desliga-se do nitrogênio de outro aminoácido. A ligação de dois aminoácidos tem como subproduto uma molécula de água (H_2O) e dois resíduos ligados pela ligação peptídica.

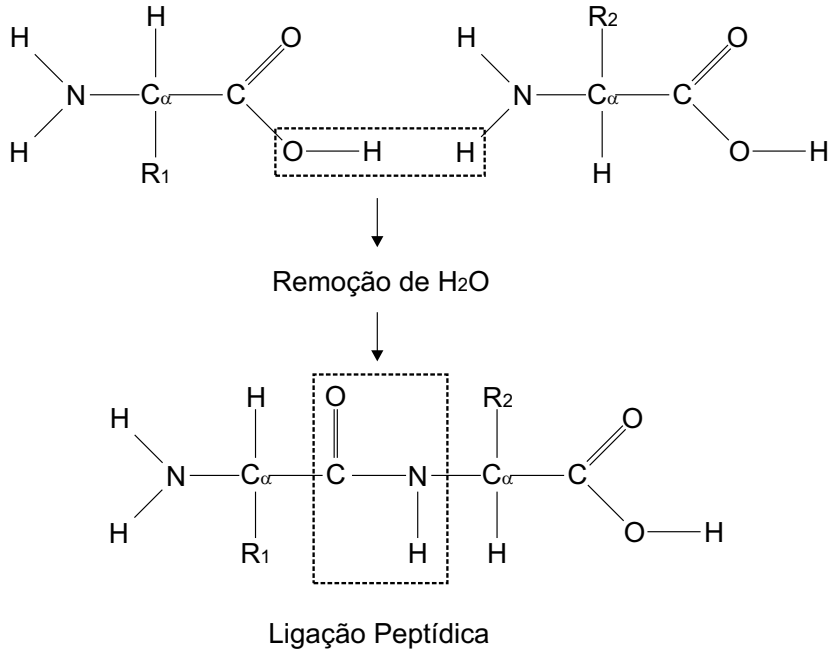


Figura 2.2: Ligação Peptídica

Diz-se então que uma proteína é composta de um conjunto de resíduos. O comprimento da cadeia de resíduos influi também na estrutura final da proteína. Na natureza, o número de resíduos contidos em uma proteína pode variar de pouco mais de 100 até mesmo mais de 5000.

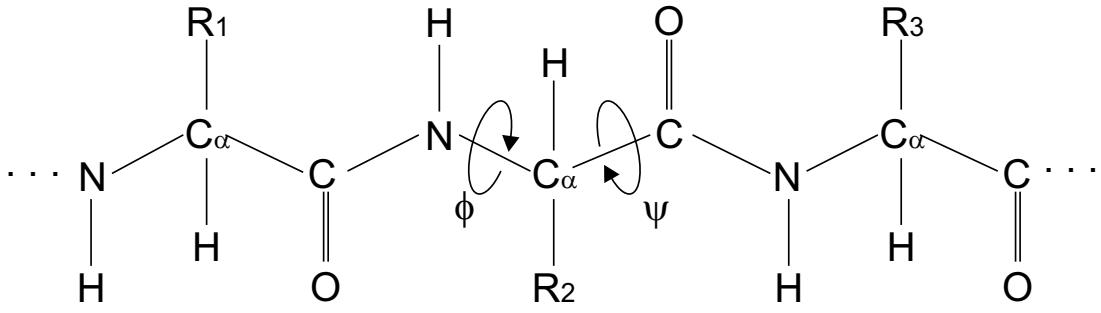


Figura 2.3: Planos rotacionais ϕ e ψ

Uma característica das proteínas é que o C_α dos seus resíduos define dois eixos de rotação. Estes eixos são chamados ϕ e ψ e são mostrados na Figura 2.3. Estes eixos de rotação são livres. Como as cadeias laterais de cada resíduo apresentam forças moleculares entre si, estas forças farão com que os resíduos

acabam girando nos eixos ϕ e ψ a fim de obter uma conformação mais adequada à energia potencial da proteína.

Esta conformação espacial dos resíduos acaba por dar origem às chamadas estruturas primária, secundária, terciária e quaternária das proteína. A função de uma proteína está intimamente ligada à estrutura espacial assumida por esta, definindo com quais outras proteínas ou moléculas esta irá se ligar com maior facilidade. Um exemplo disto é a produção de anticorpos, que devem possuir geralmente uma forma intrinsecamente relacionada à membrana plasmática da bactéria invasora.

2.1.3 DNA

O código genético, conjunto de informações que regulam praticamente todo o comportamento celular, é composto pelo DNA. O DNA, ou ácido desoxirribonucléico, é composto por uma série de elementos chamados nucleotídeos. Cada nucleotídeo é composto por um fosfato, um açúcar e uma base nitrogenada[27]. A Figura 2.4 mostra uma molécula de Desoxirribose que, no caso, é o açúcar que compõe o DNA.

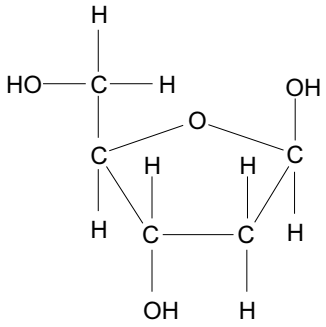


Figura 2.4: Molécula de Desoxirribose[47]

A Figura 2.5 mostra as bases nitrogenadas que compõem o DNA. São estas as purinas (Adenina e Guanina) e as pirimidinas (Timina e Citosina).

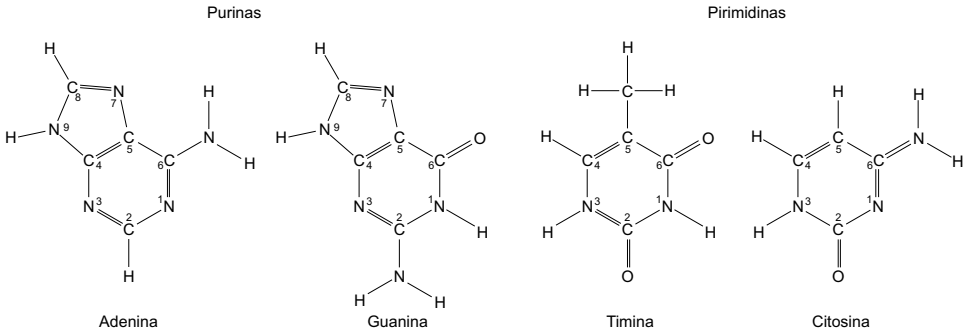


Figura 2.5: Bases Nitrogenadas

Os nucleotídeos ligam-se uns aos outros através de seus grupos fosfatos formando uma "fita" direcionada. As fitas de DNA possuem dois extremos distintos denominados 5' e 3'. Na natureza o DNA encontra-se organizado em pares destas fitas, conectadas entre si antiparalelamente. A Figura 2.6 ilustra uma fita dupla de DNA.

Em uma fita dupla de DNA a Adenina irá sempre se parear com a Timina. Já a Guanina irá sempre se parear a uma Citosina. Assim, a Adenina e a Timina são ditas bases complementares, assim como a Guanina e a Citosina. Esta relação de complementaridade é explicada pelo fato de a Adenina e a Citosina estabelecerem entre si duas pontes de hidrogênio, enquanto que a Guanina e a Citosina estabelecem sempre três pontes de hidrogênio entre si.

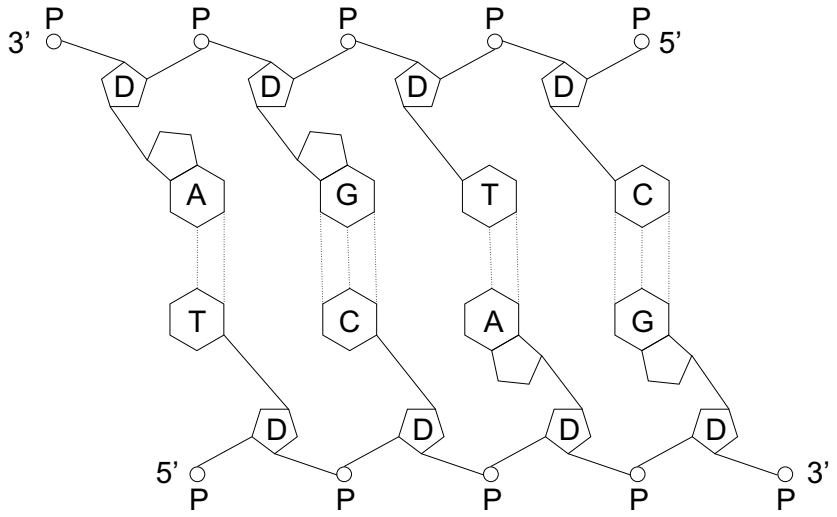


Figura 2.6: Fita dupla de DNA[47]

O comprimento de uma seqüência é medido em função do número de nucleotídeos, ou *base pairs*(bp). Os prefixos multiplicadores também são usados na indicação das medidas, como Mbp ou Kbp, indicando milhões ou milhares de nucleotídeos respectivamente.

A fita mostrada na Figura 2.6 assume ainda uma disposição de fita duplamente helicoidal. A fita helicoidal pode, durante o processo de divisão celular dos eucariontes, enrolar-se de forma a criar uma espécie de novelo em forma de "X". A tal conformação dá-se usualmente o nome de cromossomo. Já nas bactérias o DNA pode encontra-se estruturado circularmente como em um anel. Esta conformação é denominada de plasmídio. Já nos vírus, o material genético pode encontra-se até mesmo como uma fita sem macro-estrutura definida.

Nem todo material genético de uma célula tem função aparente. As regiões funcionais, responsável pela síntese de RNA, são denominadas genes. O RNA é também um ácido nucleico e tem importante papel na síntese proteica.

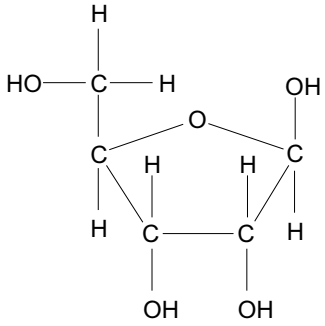


Figura 2.7: Molécula de Ribose[47]

2.1.4 RNA

O RNA, ou ácido ribonucleico, faz parte do processo de transcrição do DNA em proteínas. Semelhantemente ao DNA, o RNA possui uma molécula de açúcar, que neste caso é a Ribose. A Figura 2.7 mostra a estrutura da Ribose.

Existem três categorias de RNA: RNA mensageiro, RNA transportador e RNA ribossômico. O RNA mensageiro, ou mRNA, é uma estrutura semelhante ao DNA, mas não se apresenta como uma fita dupla, mas sim como uma fita simples de bases, ocorrendo apenas as bases Adenina, Uracila, Guanina e Citosina. Neste caso, a Uracila é a base complementar da Adenina. O mRNA é formado no interior do núcleo da célula, pareando-se com uma fita de DNA, criando assim uma fita complementar de um trecho material genético da célula. A Figura 2.8 mostra a base nitrogenada Uracila.

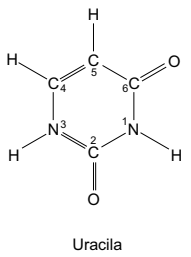


Figura 2.8: Base nitrogenada Uracila

O RNA ribossômico, ou rRNA, compõe boa parte dos Ribossomos, que são organelas presentes no citoplasma da célula. Os ribossomos são fundamentais na síntese protéica. O RNA transportador, ou tRNA, é responsável por carregar os aminoácidos dispersos no citoplasma e trazê-los para perto dos ribossomos, auxiliando assim o processo de ligação dos aminoácidos para que estes constituam as proteínas.

2.1.5 Síntese Protéica

O processo de produção do mRNA chama-se transcrição. Na presença de uma enzima chamada RNA-polimerase, um determinado trecho de uma fita de DNA

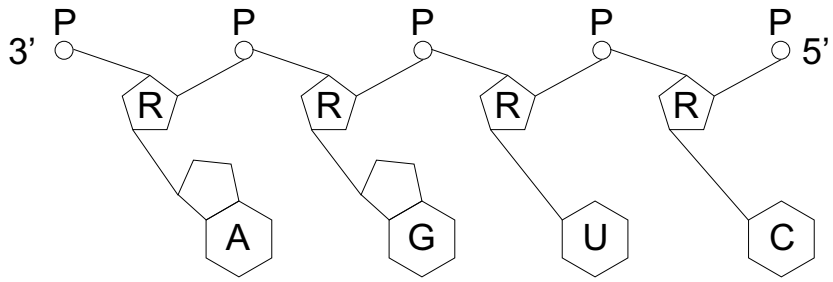


Figura 2.9: Fita de mRNA[47]

desliga-se temporariamente de seu par complementar. Bases de RNA começam então a se ligar a esta fita desconectada. É obedecida a regra na qual uma base do DNA sempre se liga à sua contraparte no RNA, observando-se que no RNA a Uracila ocorre no lugar da Timina. A fita de mRNA separa-se então da fita de DNA, e esta então volta a se ligar ao seu par.

O mRNA recém-formado contém informações necessárias para a célula sintetizar uma proteína. Esta informação encontra-se organizada em códons, nome dado a cada tripla de bases no RNA. Cada uma dessas triplas carrega uma forma de instrução que guiará a síntese protéica. É interessante notar que a informação contida no mRNA depende não somente da fita de DNA da qual este provém, mas também em que ponto desta será será iniciado o processo de transcrição.

```
3' ATGCTAGCTGATCGATCTGAC 5'
5' TACGATCGACTAGCTAGACTG 3'
```

Figura 2.10: Exemplo de seqüência de DNA e extremidades 5' e 3'

Por exemplo, na seqüência da Figura 2.10, a transcrição pode ser iniciada na primeira base da fita de baixo, designando os códons: TAC, GAT, CGA, etc... A transcrição pode iniciar-se também a partir da segunda ou da terceira base, originando os seguintes códons respectivamente: ACG, ATC, GAC, etc... e CGA, TCG, ACT, etc... A transcrição também pode ocorrer da mesma forma da segunda fita de DNA, dando origem assim a seis possíveis interpretações para cada fita de DNA.

Vale lembrar que essa possibilidade existe e efetivamente ocorre na natureza. O mecanismo que regula os trechos a serem transcritos utiliza os chamados promotores, códons de iniciação e os códons de parada, responsáveis por indicar o ponto de início da transcrição, o ponto de início do gene e de término de um gene respectivamente.

Uma vez terminada a transcrição, o mRNA irá se deslocar do núcleo para o citoplasma, onde estão os ribossomos. Ocorrerá então o *splicing* do mRNA, que é a retirada de trechos do mRNA que não são responsáveis pela codificação de proteínas. O splicing é regulado por determinados códons que são chamados de códons de *splicing*, responsáveis por delimitar os *íntrons*, como são chamados estes trechos a serem removidos do mRNA. Os trechos que sobram são chamados de *éxons* e carregam os códons a serem efetivamente traduzidos em um proteína.

Segue-se então o processo de tradução do mRNA no ribossomo. Este é responsável por "ler" o que sobrou da fita de mRNA após o *splicing* e ir agregando uma seqüência de aminoácidos. Os aminoácidos a serem agregregados são definidos pela seqüência de códons do mRNA. Os aminoácidos são carregados pelo tRNA, que liga-se à fita de mRNA e faz com que os aminoácidos possam dispor-se próximo o suficiente entre si para se ligarem. Cada códon define um aminoácido a ser agregado à nova proteína sendo sintetizada.

<table>
<tr><th>1ª base</th><th colspan="4">2ª base</th><th>3ª base</th></tr>
<tr><td></td><td>U</td><td>C</td><td>A</td><td>G</td><td></td></tr>
<tr><td rowspan="4">U</td><td rowspan="2">Phe(F)*</td><td rowspan="3">Ser(S)%</td><td rowspan="2">Tyr(Y)%</td><td rowspan="2">Cys(C)%</td><td>U</td></tr>
<tr><td>C</td></tr>
<tr><td rowspan="2">Leu(L)*</td><td rowspan="2">TERM</td><td rowspan="2">TERM</td><td>A</td></tr>
<tr><td>G</td></tr>
<tr><td rowspan="4">C</td><td rowspan="4">Leu(L)*</td><td rowspan="4">Pro(P)*</td><td>His(H)+</td><td rowspan="4">Arg(R)+</td><td>U</td></tr>
<tr><td>C</td></tr>
<tr><td>Gln(Q)%</td><td>A</td></tr>
<tr><td>G</td></tr>
<tr><td rowspan="4">A</td><td rowspan="2">Ile(I)*</td><td rowspan="4">Thr(T)%</td><td>Asn(N)%</td><td>Ser(S)%</td><td>U</td></tr>
<tr><td>C</td></tr>
<tr><td>Lys(K)+</td><td>Arg(R)+</td><td>A</td></tr>
<tr><td>Met(M)*</td><td>G</td></tr>
<tr><td rowspan="4">G</td><td rowspan="4">Val(V)</td><td rowspan="4">Ala(A)*</td><td>Asp(D)-</td><td rowspan="4">Gly(G)*</td><td>U</td></tr>
<tr><td>C</td></tr>
<tr><td>Glu(E)-</td><td>A</td></tr>
<tr><td>G</td></tr>
</table>

Tabela 2.2: Código Genético. Aminoácidos em suas representações de três e uma letra respectivamente.

Como cada códon é representado por seqüência de 3 bases, dentre os 4 tipos de bases disponíveis, isso permitiria condificar $4^3 = 64$ aminoácidos diferentes. No entanto, há apenas 20 aminoácidos ocorrendo livremente na natureza. A associação entre códons e aminoácidos está presente na Tabela 2.2, onde os códons de parada são indicados por TERM, aminoácidos apolares são indicados por (*), polares não carregados por (%), polares com carga positiva por (+) e polares com carga negativa por (-).

Desta forma, há uma redundância no código genético, pois a terceira posição de cada códon muitas vezes é indiferente no processo de tradução. Supõe-se que esta redundância seja capaz de proteger, até certo ponto, uma seqüência genética do fenômeno do poliformismo de base única (SNP), onde apenas uma base da seqüência é alterada.

Nem todo material genético de um organismo é responsável por sintetizar proteínas. Sabe-se que uma determinada parcela do código genético é responsável por manter a sua própria integridade, dando-se a este código a denominação de *housekeeping*. Ao trecho de código genético responsável pela síntese de RNA, dá-

Característica	DNA	RNA
Pentose	Desoxirribose	Ribose
Bases púricas	Adenina e Guanina	Adenina e Guanina
Bases pirimídicas	Citosina e Timina	Citosina e Uracila
Estrutura	Duas cadeias helicoidais	Uma cadeia
Enzima hidrolítica	Desoxirribonuclease	Ribonuclease
Origem	Replicação	Transcrição
Enzima sintética	DNA-polimerase	RNA-polimerase
Função	Informação genética	Síntese de proteína

Tabela 2.3: Diferenças entre o DNA e o RNA

se o nome de gene[31]. Estima-se hoje em dia que apenas 3% do código genético do ser humano seja composto de genes. Mesmo sendo uma parcela muito pequena, esta é capaz de sintetizar uma grande diversidade de proteínas quando comparada com a diversidade proteica de outros organismos. A complexidade do organismo do ser humano é fruto dessa grande diversidade de proteínas sintetizadas.

A Tabela 2.3 apresenta um resumo das principais diferenças entre o DNA e o RNA.

2.2 Comparação e alinhamento entre seqüências genéticas

Para se compreender melhor a função de cada gene, são necessárias ferramentas capazes de determinar a presença e localização deste dentro do material genético de um organismo. A localização de genes envolve, geralmente, procurar por uma informação já conhecida. Parte-se de uma seqüência conhecida e deseja-se saber se esta seqüência está presente, de alguma forma, no código genético de outro organismos. Tal procedimento é chamado de *alinhamento*, consistindo basicamente em parear uma seqüência com a outra, comparando seus nucleotídeos a fim de descobrir qual seria o melhor pareamento entre as duas seqüências.

Além disso, deseja-se obter uma medida que indique a semelhança entre seqüências genéticas de dois ou mais organismos de uma espécie, ou até mesmo de espécies diferentes. Neste caso, estamos interessados em descobrir qual é a *similaridade*[47] entre seqüências genéticas. Para ilustrar o conceito de similaridade, a Figura 2.11 mostra uma situação onde duas seqüências de DNA foram alinhadas e suas bases são comparadas.

Para cada par de bases iguais, uma valor é somado à pontuação final, no caso +1. Caso as bases comparadas sejam diferentes, então uma penalidade é atribuída à pontuação total, no caso -1. Para obter melhores pontuações, é possível adicionarmos espaços (*gaps*) nas seqüências de DNA. Tal artifício é compatível com o processo evolutivo das seqüências genéticas e corresponderia à representação de deleções ou inserções de novas bases. Para fins de pontuação, tal situação contribui com um desconto maior , no caso ilustrado -2.

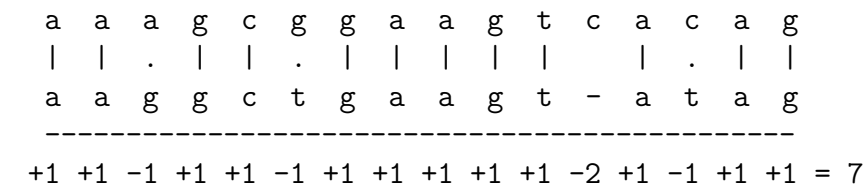


Figura 2.11: Exemplo de obtenção de um valor de similaridade. É mostrado um alinhamento entre as seqüências "aagcggaagtcacag" e "aaggctgaagtatag"

O alinhamento mostrado na Figura 2.11 pode não ser aquele que produz a melhor similaridade. Existem dois métodos básicos para comparação de seqüências e obtenção dos melhores alinhamentos. Estes serão descritos a seguir.

2.3 Alinhamento global

O processo evolutivo dos seres vivos consiste, em boa parte, na modificação e diversificação de seu material genético. Alguns dos eventos que colaboram para tais modificações são as deleções e inserções, como foi citado em 2.2.

Uma deleção é o resultado de uma mutação onde um nucleotídeo (ou uma seqüência destes) é removido da seqüência genética, originando uma seqüência menor. O contrário também ocorre, quando uma célula é infectada por um vírus. Neste exemplo, o vírus introduz parte do seu material genético no genoma da célula hospedeira.

Quando há interesse em descobrir a difereça existente entre duas seqüências bem semelhantes, o método do Alinhamento Global é utilizado. O algoritmo exato que realiza tal tarefa é o de Needleman-Wunsch [39]. Tal algoritmo é capaz de, dadas duas seqüências, obter a melhor pontuação de similaridade e ainda apresentar quais foram os alinhamentos que produziram tal pontuação.

Como foi mostrado na Figura 2.11, espaços podem ser adicionados em pontos arbitrários de uma das seqüências para que a pontuação final do alinhamento possa ser maior. Tais espaços são denominados *gaps*. A única limitação neste caso é que um *gap* inserido em uma sequência não pode estar pareado com um *gap* inserido na outra seqüência. Finalizado o processo de alinhamento global, ambas as seqüências possuirão o mesmo comprimento, considerando-se os *gaps* adicionados.

O algoritmo de Needleman-Wunsch é baseado no paradigma de programação dinâmica [10]. A programação dinâmica utiliza-se de uma técnica chamada memorização, acelerando a obtenção de valores para certos tipos de equações de recorrência, onde o resultado de um problema depende de instâncias menores do mesmo problema.

O algoritmo pode ser dividido em duas fases. Na primeira fase, é feita a obtenção da matriz de similaridades. Na segunda fase, são produzidos e exibidos os alinhamentos entre as seqüências comparadas.

A idéia da primeira fase é, dadas duas seqüências de nucleotídeos S e T , obter os valores de similaridade $sim(S[1 \ldots i], T[1 \ldots j])$, para $1 \leq i \leq |S|$ e $1 \leq j \leq |T|$,

onde $|S|$ e $|T|$ indicam o comprimento das seqüências S e T, enquanto que $S[1 \ldots i]$ representa o prefixo formado pelos i primeiros caracteres da seqüência S. A Equação 2.1 mostra como são obtidos os valores de similaridade:

$$sim(S\,[1 \ldots i]\,, T\,[1 \ldots j]) = max \begin{cases} sim(S[1 \ldots i], T[1 \ldots j-1]) - g \\ sim(S[1 \ldots i-1], T[1 \ldots j-1]) + p(i,j) \\ sim(S[1 \ldots i-1], T[1 \ldots j]) - g \end{cases} \tag{2.1}$$

Onde $p(i,j)$ equivale a um valor positivo, se $s_i = t_j$, o que indica uma semelhança (*match*). Caso contrário, $p(i,j)$ possui um valor negativo (*mismatch*). Os outros casos representam a situação quando uma base de S ou de T é alinhada com um *gap*. O valor g indica qual a penalidade associada ao fato de se introduzir um *gap* no alinhamento. A penalidade associada a um *gap* é geralmente maior que a penalidade de um *mismatch*. A Equação 2.1 é claramente recursiva.

O método da programação dinâmica, neste caso, consiste em armazenar os resultados intermediários obtidos em uma matriz de similaridades $\mathrm{S}_{m \times n}$. Considerando que um caractere '-' ,representando um *gap*, é prefixado a S e a T, temos que a matriz de similaridades tem as dimensões $(|S|+1) \times (|T|+1)$, onde cada elemento $\mathrm{S}_{i,j}$ da matriz possui o valor de $sim(S\,[1 \ldots i]\,, T\,[1 \ldots j])$.

Antes iniciar a obtenção dos valores de similaridade, a matriz deve ser inicializada. Isto é feito atribuindo-se o valor $i \times g$ a todas as células $\mathrm{S}_{i,0}$ e $j \times g$ a todas as células $S_{0,j}$. Esta inicialização contabiliza a existência de *gaps* inseridos no início do alinhamento.

A utilização da programação dinâmica consegue agilizar enormemente a obtenção dos valores da similaridade entre S e T. No entanto, tal abordagem introduz uma dependência muito restritiva entre os elementos $\mathrm{S}_{i,j}$ da matriz de similaridades. Cada valor $\mathrm{S}_{i,j}$ só pode ser calculado se os valores $\mathrm{S}_{i-1,j}$, $\mathrm{S}_{i-1,j-1}$ e $\mathrm{S}_{i,j-1}$ também já tiverem sido calculados. A Figura 2.12 apresenta o algoritmo de preenchimento dos valores da matriz de similaridades.

Uma vez obtidos os valores da matriz de similaridades, no elemento $\mathrm{S}_{m+1,n+1}$ encontra-se a pontuação do melhor alinhamento entre S e T. Além disso é possível obter exatamente o conjunto de alinhamentos que deram origem a tal pontuação. Note que é possível obter mais de um alinhamento com pontuação ótima, devido à liberdade oferecida pelos *gaps* inseridos durante o alinhamento. Waterman[55] mostrou que, para duas seqüências de tamanho n, há aproximadamente $(1+\sqrt{2})^{2n+1} n^{-\frac{1}{2}}$ alinhamentos globais ótimos possíveis.

De posse da matriz de alinhamentos preenchida, o algoritmo descrito na Figura 2.13 exibe o melhor alinhamento dentre os obtidos. Uma vez que vários alinhamentos podem produzir o mesmo valor de similaridade, adota-se aqui o melhor alinhamento como sendo aquele que, em primeiro caso, originou-se do elemento $\mathrm{S}_{i-1,j}$ superior a $\mathrm{S}_{i,j}$, em segundo caso, do elemento $\mathrm{S}_{i-1,j-1}$, e em terceiro caso, do elemento $\mathrm{S}_{i,j-1}$.

O algoritmo de Needleman-Wunsch possui complexidade de tempo $O(mn)$, onde $m = |S|$ e $n = |T|$, o que para seqüências de tamanho semelhante traduz-se em $O(n^2)$. O mesmo também é válido no que diz respeito à complexidade de espaço, pois a matriz bi-dimensional $(m+1) \times (n+1)$ também possui tamanho

Algoritmo Similaridade

Entrada:Seqüências S e T
Saída: Matriz de similaridade $\mathrm{S}_{m+1\times n+1}$ entre S e T
$m \leftarrow |s|$
$n \leftarrow |t|$
for $i \leftarrow 0$ **to** m **do**
    $\mathrm{S}_{i,0} \leftarrow i \times g$
for $j \leftarrow 0$ **to** n **do**
    $\mathrm{S}_{0,j} \leftarrow j \times g$
for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** n **do**
        $\mathrm{S}_{i,j} \leftarrow max[\mathrm{S}_{i-1,j} + g$
                $\mathrm{S}_{i-1,j-1} + p(i,j)$
                $\mathrm{S}_{i,j-1} + g]$
return $\mathrm{S}_{m,n}$

Figura 2.12: 1º passo do algoritmo de Needleman-Wunsch

Algoritmo Alinhamentos

Entrada: Índices i, j, matriz S preenchida
Saída: Seqüências S', T', e comprimento len
if $i = 0$ and $j = 0$ **then**
    $len \leftarrow 0$
else if $i > 0$ and $\mathrm{S}_{i,j} = \mathrm{S}_{i-1,j} + g$ **then**
    Align$(i-1, j, len)$
    $len \leftarrow len + 1$
    $s'_{len} \leftarrow s_i$
    $t'_{len} \leftarrow -$
else if $i > 0$ and $j > 0$ and $\mathrm{S}_{i,j} = \mathrm{S}_{i-1,j-1} + p(i,j)$ **then**
    Align$(i-1, j-1, len)$
    $len \leftarrow len + 1$
    $s'_{len} \leftarrow s_i$
    $t'_{len} \leftarrow t_j$
else //$j > 0$ and $\mathrm{S}_{i,j} = \mathrm{S}_{i,j-1} + g$
    Align$(i, j-1, len)$
    $len \leftarrow len + 1$
    $s'_{len} \leftarrow -$
    $t'_{len} \leftarrow t_j$

Figura 2.13: 2º passo do algoritmo de Needleman-Wunsch

$O(n^2)$, para seqüências de tamanho semelhante. De posse da matriz de similaridades, a obtenção dos alinhamentos pode ser realizada em tempo $O(n)$ e espaço $O(n)$.

Na Figura 2.14 é mostrada a matriz de similaridades para o alinhamento de $AGTC$ e $AGTAC$ produzida pelo algoritmo de Needleman-Wunsch.

	-	**A**	**G**	**T**	**A**	**C**
-	0	←-2	←-4	←-6	←-8	←-10
A	↑-2	↖1	←-1	←-3	←↖-5	←-7
G	↑-4	↑-1	↖2	←0	←-2	←-4
T	↑-6	↑-3	↑0	↖3	←1	←-1
C	↑-8	↑-5	↑-2	↑1	↖2	↖2

Figura 2.14: Matriz de similaridades $S_{i,j}$ produzida pelo algoritmo de Needleman-Wunsch para o alinhamento entre $AGTC$ e $AGTCA$ para *match*=1, *mismatch*=-1 e *gap*=-2. São indicados também a partir de onde os valores das células foram obtidos.

2.4 Alinhamento local

O alinhamento global produz ótimos resultados para seqüências semelhantes. Pode-se desejar, no entanto, comparar seqüências essencialmente distintas que possuam alguns segmentos muito semelhantes.

Quando se deseja procurar por tais segmentos, o algoritmo mais indicado é o de Smith-Waterman[49], capaz de realizar o alinhamento local de seqüências. No alinhamento local, procura-se descobrir quais são as subseqüências mais parecidas entre duas seqüências genéticas.

O algoritmo de Smith-Waterman assemelha-se muito ao algoritmo de alinhamento global, possuindo algumas diferenças. Tais diferenças partem da idéia de que não serão penalizados os *gaps* inseridos no início e no final das seqüências. Além disso, sabendo-se que as seqüências apresentam diferenças significativas, a perda de pontuação proveniente de diferenças entre as seqüências será computada de forma mais branda.

Essas diferenças traduzem-se efetivamente em duas modificações. A primeira, quando se dá a inicialização da primeira linha e da primeira coluna da matriz de similaridades. Assim, $\mathrm{S}_{i,0} = \mathrm{S}_{0,j} = 0$ para $1 \leq i \leq |S|$ e $1 \leq j \leq |T|$. Isto significa que um alinhamento pode possuir um número arbitrário de *gaps* em seu início. A outra modificação é na obtenção dos valores $\mathrm{S}_{i,j}$. Estes passam a ser obtidos como é mostrado na Equação 2.2

$$sim(S\,[1\ldots i]\,,T\,[1\ldots j]) = max\begin{cases} sim(S[1\ldots i], T[1\ldots j-1]) - g \\ sim(S[1\ldots i-1], T[1\ldots j-1]) + p(i,j) \\ sim(S[1\ldots i-1], T[1\ldots j]) - g \\ 0 \end{cases} \tag{2.2}$$

O valor 0 inserido na Equação 2.2 impede que a pontuação dos alinhamentos seja menor que 0. Isto permite que, quando subseqüências muito semelhantes forem encontradas, as diferenças encontradas anteriormente não prejudiquem a pontuação a ponto de impedir a identificação destes alinhamentos locais.

O melhor alinhamento local poderá ser encontrado a partir da entrada $S_{i,j}$ de maior valor. Para obter o melhor alinhamento local, basta aplicar o algoritmo de obtenção de alinhamentos (Figura 2.13), a partir da entrada $S_{i,j}$. Neste caso, o algoritmo irá parar não mais quando atingir $S_{1,1}$, mas sim quando atingir uma entrada na tabela com o valor 0.

Em geral interessa não somente o melhor alinhamento local, mas sim aqueles que obtiveram uma pontuação acima de um limiar estabelecido. Isso significa que o algoritmo de Smith-Waterman irá produzir vários alinhamentos locais.

O algoritmo de Smith-Waterman, assim como o de Needleman-Wunsch, também apresenta complexidade de tempo e de espaço $O(n^2)$, onde n é o tamanho das seqüências comparadas.

Na Figura 2.15 é mostrada a matriz de similaridades para o alinhamento de $AGTC$ e $AGTAC$ produzida pelo algoritmo de Smith-Waterman.

	-	**A**	**G**	**T**	**A**	**C**
-	0	0	0	0	0	0
A	0	↖1	0	0	↖1	0
G	0	0	↖2	0	0	0
T	0	0	0	↖3	←1	0
C	0	0	0	↑1	↖2	↖2

Figura 2.15: Matriz de similaridades $S_{i,j}$ produzida pelo algoritmo de Smith-Waterman para o alinhamento entre $AGTC$ e $AGTCA$, para *match*=1, *mismatch*=-1 e *gap*=-2. São indicados também a partir de onde os valores das células foram obtidos.

2.5 *Gaps*

Os algoritmos mostrados em 2.3 e 2.4 atribuem um valor constante a *gaps*, independentemente da disposição dos mesmos no alinhamento. Tal avaliação dada aos *gaps* inseridos em um alinhamento não corresponde geralmente com os processos mutagênicos sofridos por um material genético.

Eventos, como a deleção e a inserção, podem remover ou adicionar uma seqüência inteira de bases em um único evento mutacional. Além disso, é muito

mais provável a ocorrência de um evento de deleção ou de inserção de várias bases do que vários eventos de deleção ou inserção de uma só base. Assim, para corresponder à realidade, deve-se considerar então que os *gaps* tendem a ocorrer em seqüência.

Para priorizar a ocorrência de gaps consecutivos, define-se $w(k)$, para $k \geq 1$, como a função de *gaps* que determina o valor de k *gaps* inseridos consecutivamente. Os algoritmos vistos até o momento utilizam uma função constante de *gaps* $w(k) = gk$, onde g equivale ao valor atribuído aos *gaps* em um alinhamento.

O método da programação dinâmica pode ser adaptado para alinhar seqüências levando-se em conta uma função genérica de *gaps* $w(k)$. Um algoritmo apresentado por Smith e Waterman[56] é capaz de trabalhar com tais funções genéricas de gaps, porém apresenta como desvantagem complexidade de tempo $O(n^3)$.

Tal limite de tempo torna inviável a utilização deste algoritmo para a obtenção dos valores de alinhamento utilizando-se funções genéricas de *gaps* para seqüências grandes. Ainda assim, é possível utilizar-se uma classe de funções menos genérica que produz alinhamentos que correspondem mais à realidade. Desta forma, precisamos obter uma função $w(k)$ que satisfaça à equação 2.3

$$w(k) \leq kw(1) \tag{2.3}$$

Que de uma forma mais geral corresponde à equação 2.4

$$w(k_1 + k_2 + \ldots + k_n) \leq w(k_1) + w(k_2) + \ldots + w(k_n) \tag{2.4}$$

Se $w(k)$ satisfaz à equação 2.4, então esta é chamada de *função sub-aditiva*. Tomando-se então uma função discreta w da forma $w(k) = uk + v$, $k \geq 1$. Se $u, v > 0$, então $w(k)$ é uma função sub-aditiva. Em outras palavras, usar uma função discreta permite atribuir um custo maior à "abertura" de *gaps* do que à extensão dos mesmos, priorizando alinhamentos que possuam *gaps* contíguos.

2.5.1 Gotoh

Baseando-se nestas premissas, Gotoh [22] adaptou os algoritmos existentes em programação dinâmica para produzir um alinhamento usando funções discretas para as penalidades de *gaps*. Tal algoritmo possui complexidade de tempo $O(n^2)$.

O algoritmo de Waterman[56] gera uma matriz de distâncias de edição $\mathrm{D}_{i,j}$ baseada na seguinte indução:

$$\mathrm{D}_{i,j} = min[\mathrm{S}_{i-1,j-1} + p(i,j), \mathrm{P}_{i,j}, \mathrm{Q}_{i,j}] \tag{2.5}$$

onde

$$\mathrm{P}_{i,j} = min_{1 \leq k \leq i} \left[S_{i-k,j} + w(k)\right] \tag{2.6}$$

e

$$\mathrm{Q}_{i,j} = min_{1 \leq k \leq j} \left[S_{i,j-k} + w(k)\right] \tag{2.7}$$

Em 2.6 e 2.7, $\mathrm{P}_{i,j}$ e $\mathrm{Q}_{i,j}$ são obtidos em $i-1$ e $j-1$ passos.

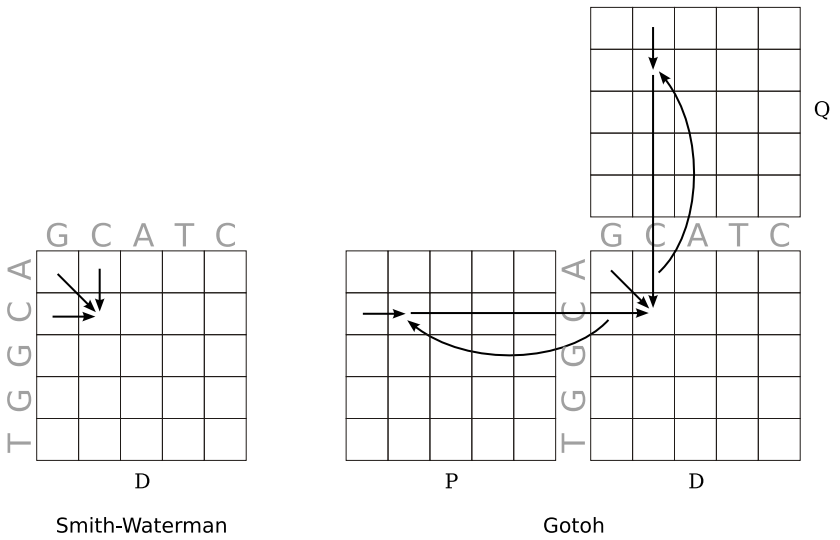


Figura 2.16: Relações de dependência entre D, P e Q. No algoritmo de Smith-Waterman, cada valor de $D_{i,j}$ depende de outros três. No algoritmo de Gotoh, além de cada valor de $D_{i,j}$ depender de outros três, cada valor de $P_{i,j}$ e de $Q_{i,j}$ depende de outros dois valores.

Gotoh utiliza um função linear de *gaps* $w(k) = uk+v$ onde v é o valor atribuído à "abertura" de um *gap* e u é valor atribuído a uma "extensão"de *gaps*. Com a função linear $w(k)$, é possível calcular $P_{i,j}$ e $Q_{i,j}$ em tempo constante, de acordo com as relações 2.8 e 2.9.

$$\begin{aligned} P_{i,j} &= min\left[D_{i-1,j} + w(1), min_{2\leq k\leq i}\left(D_{i-k,j} + w(k)\right)\right] \\ &= min\left[D_{i-1,j} + w(1), min_{1\leq k\leq i-1}\left(D_{i-1-k,j} + w(k+1)\right)\right] \\ &= min\left[D_{i-1,j} + w(1), min_{1\leq k\leq i-1}\left(D_{i-1-k,j} + w(k) + u\right)\right] \\ &= min\left[D_{i-1,j} + w(1), P_{i-1,j} + u\right] \qquad (2.8) \end{aligned}$$

$$e$$

$$Q_{i,j} = min\left[D_{i,j-1} + w(1), Q_{i,j-1} + u\right] \qquad (2.9)$$

Assim, a produção dos valores de $D_{i,j}$ pode ser completada em tempo $O(mn)$, sendo necessário para cada valor de $D_{i,j}$ escolher o menor de três valores e, para $P_{i,j}$ e $Q_{i,j}$ escolher o menor de dois valores. A Figura 2.16 ilustra esta relação entre os valores citados.

O algoritmo de Gotoh pode ser utilizado para o produção de alinhamentos globais e alinhamentos locais. Para alinhamentos globais, $D_{i,0} = P_{i,0} = w(j)$, para $1 \leq i \leq m$, e $D_{0,j} = Q_{0,j} = w(j)$, para $1 \leq j \leq N$. Já para a obtenção de alinhamentos locais, $D_{i,0} = P_{i,0} = 0$ e $D_{0,j} = Q_{0,j} = 0$.

A produção de alinhamentos é feita pela utilização de uma matriz de *traceback* $e_{i,j}$, cujos elementos são números de três *bits* indicando as direções de onde o valor $D_{i,j}$ se originou.

2.6 Aprimoramentos sobre os algoritmos de comparação de seqüências

2.6.1 Reduzindo o custo de memória

Quando se leva em conta que o comprimento de uma cadeia de apenas um cromossomo humano possui aproximadamente vários milhões de pares de bases, percebe-se que torna-se proibitivo o custo quadrático de memória imposto pelos algoritmos mostrados em 2.3, 2.4 e 2.5.

Ainda assim, é possível produzir um alinhamento e seu *score* através do método de programação dinâmica utilizando-se um espaço linear em função das seqüências sendo comparadas. Quando se deseja apenas obter o valor final obtido pelo alinhamento, basta notar que cada linha da matriz de similaridades depende apenas da linha anterior. Desta forma, basta manter na memória apenas a linha sendo calculada e a sua predecessora.

2.6.1.1 Hirschberg

Myers e Miller [38] perceberam uma aplicação direta do algoritmo de Hirschberg [24] para a obtenção de subseqüências comuns maximais (*Longest Common Subsequences*) ao problema de comparação de seqüências genéticas.

O ponto chave do algoritmo é definir um determinado i, onde $1 \leq i \leq m$, e descobrir para este i, qual o valor de j tal que $a(i,j) = max(a(i, 0 \ldots n))$. Em outras palavras, significa escolher uma linha da matriz S de similaridades, e para esta linha, encontrar o valor de j que corresponde ao ponto onde o alinhamento ótimo intercepta a linha i. Neste ponto, há duas possibilidades:

1. s_i é alinhado com t_j, para $1 \leq j \leq n$
2. s_i é alinhado com um *gap* entre t_j e t_{j+1}, para $0 \leq j \leq n$

Definindo

$$Alin\left(\frac{X}{Y}\right) \tag{2.10}$$

como sendo o alinhamento ótimo entre as seqüências X e Y, os alinhamentos indicados nos casos (1) e (2) podem ser descritos respectivamente como

$$Alin\begin{pmatrix} S[1 \ldots i-1] \\ T[1 \ldots j-1] \end{pmatrix} + \begin{matrix} s_i \\ t_j \end{matrix} + Alin\begin{pmatrix} S[1 \ldots i-1] \\ T[1 \ldots j-1] \end{pmatrix} \tag{2.11}$$

e

$$Alin\begin{pmatrix} S[1 \ldots i-1] \\ T[1 \ldots j] \end{pmatrix} + \begin{matrix} s_i \\ - \end{matrix} + Alin\begin{pmatrix} S[1 \ldots i-1] \\ T[1 \ldots j-1] \end{pmatrix} \tag{2.12}$$

temos assim uma relação recorrente que define o alinhamento ótimo entre duas seqüências. Desta forma, para o i escolhido, é necessário obter as similaridades entre $S[1 \ldots i-1]$ e um prefixo arbitrário de T, assim como as similaridades entre $S[i+1 \ldots m]$ e um sufixo arbitrário de T.

O método de redução do custo de espaço descrito em 2.6.1 tem como saída um vetor com os valores de todos os alinhamentos de uma seqüência com os prefixos de outra seqüência. O mesmo algoritmo, quando recebe como entrada as seqüências invertidas S^{inv} e T^{inv}, produz também um vetor, só que agora indicando todos os alinhamentos de S com os sufixos de T.

A melhor escolha para i é o valor mais próximo do metade de $|S|$. Hirschberg mostrou que tal escolha leva o algoritmo a um tempo de execução que é apenas o dobro de tempo de execução dos algoritmos tradicionais. A Figura 2.17 ilustra o algoritmo de Hirschberg.

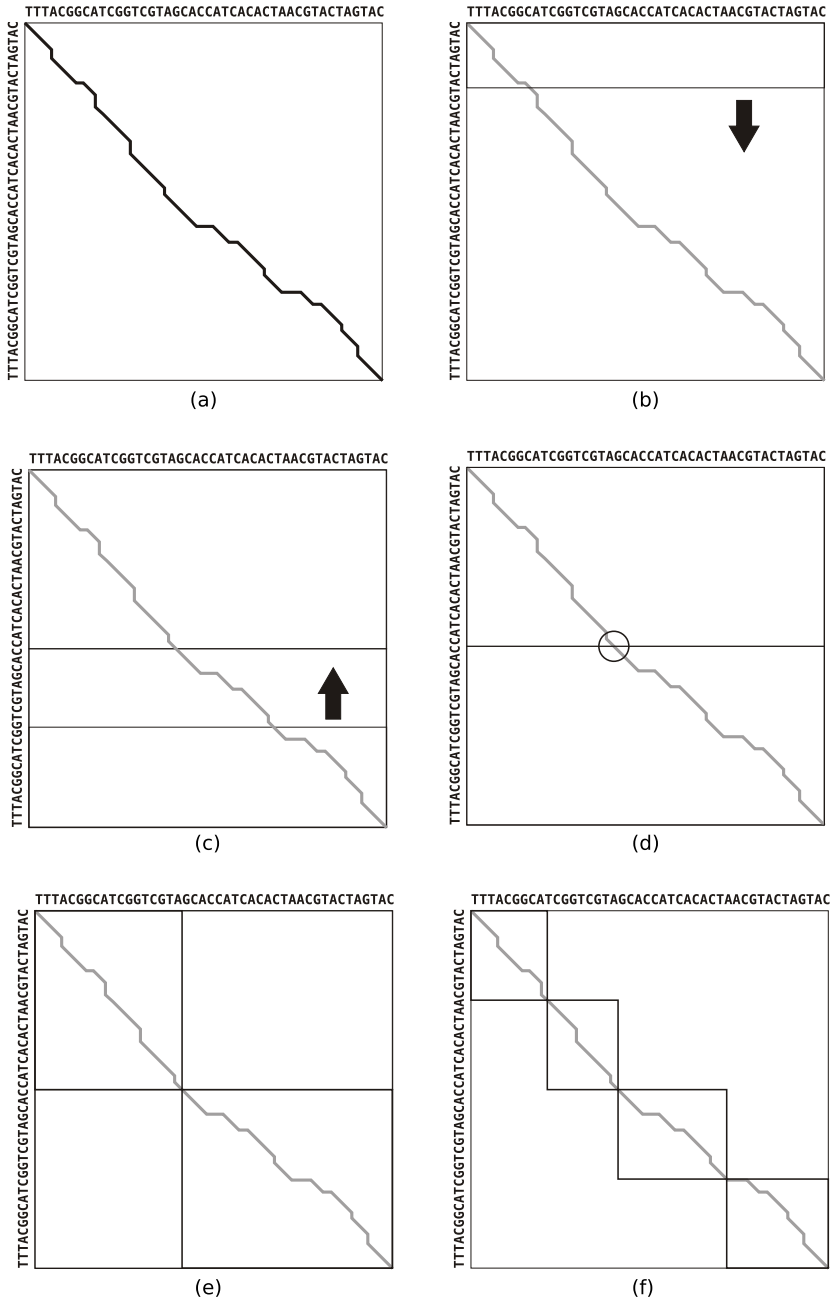


Figura 2.17: Processamento do algoritmo de Hirschberg. (a) Matriz de similaridades mostrando o alinhamento ótimo a ser obtido, desconhecido inicialmente. (b) Matriz sendo processada no sentido normal até a metade de S. (c) Matriz sendo processada no sentido inverso de S e T, também até a metade de S. (d) Encontrado o ponto onde o alinhamento ótimo cruza a linha que divide a matriz ao meio. (e) 1ª recursão do algoritmo. (f) 2ª recursão do algoritmo.

2.6.2 Reduzindo o custo de tempo

2.6.2.1 Fickett

Sob circunstâncias especiais, é possível reduzir consideravelmente o tempo necessário para se obter o alinhamento de duas seqüências S e T. Fickett [15] mostrou que, para seqüências semelhantes, é possível obter o alinhamento global em tempo $O(kn)$, onde k está diretamente relacionado à quantidade de *gaps* presentes no alinhamento global ótimo entre S e T. Isto pode ser extremamente vantajoso quando k é pequeno em comparação a $n = |S|$.

Os algoritmos de programação dinâmica para alinhamento global de seqüências geralmente preenchem toda a matriz de similaridades de cima para baixo e da esquerda para direita. Fickett mostra como re-ordenar a obtenção destes valores de forma a evitar o cálculo de valores da matriz que não contribuem para a obtenção dos alinhamentos globais ótimos.

Para tanto, o algoritmo precisa de uma estimativa d do valor do alinhamento global ótimo. Esta estimativa poderia ser obtida a partir de outros algoritmos rápidos para a obtenção de valores sub-ótimos a fim de encontrar uma estimativa superior d e, a partir desta estimativa, calcular apenas os valores $\mathrm{D}_{i,j} \leq d$, onde $\mathrm{D}_{i,j}$ representa a matriz de distâncias de edição. Às vezes já se possui uma idéia do valor para a estimativa d ou até mesmo deseja-se obter alinhamentos que possuam uma distância de edição menor que d. Outra forma, é estimar um d_0 e processar todos $\mathrm{D}_{i,j} \leq d_0$. Se o algoritmo não obtiver um alinhamento desejado, estima-se um $d_1 > d_0$ e calcula-se agora todos $\mathrm{D}_{i,j} \leq d_1$. Este processamento continuaria até que fosse encontrado um alinhamento desejado.

O cálculo dos valores $\mathrm{D}_{i,j}$ é feito então da seguinte forma. Obtém-se primeiro os valores $D_{1,1}, \ldots, D_{1,l_1}$, onde l_1 é o valor a partir do qual $D_{1,l_1} \geq d$. Assim, para $D_{1,j} > d$ para $j > l_1$. Calcula-se então $D_{2,1}, \ldots, D_{2,l_2}$, onde $D_{2,l_2} \geq d$ e $l_2 > l_1$.

Os valores vão sendo calculados desta forma até que uma linha terá o seu primeiro valor $> d$. A partir daí, valores no início e no final da linha podem ser ignorados. Desta forma, se para a i-ésima linha foram calculados os valores $D_{i,k_i}, \ldots, D_{i,l_i}$, então a linha $i+1$ será iniciada a partir de $D_{i+1,k_{i+1}}$, onde k_{i+1} é o menor valor para que $d_{i,k_{i+1}} < d$. Assim, são calculados todos os valores $\mathrm{D}_{i,j} < d$ que podem dar origem ao alinhamento ótimo.

Vale ressaltar que os valores D_{i,k_i} e D_{i,l_i} são obtidos ignorando-se os valores presentes em D_{i,k_i-1} e D_{i-1,l_i}, respectivamente. De posse dos valores $\mathrm{D}_{i,j} \leq d$, o procedimento de *traceback* é realizado para dar origem aos alinhamentos desejados.

Fickett mostra que os valores de k e l definem uma região na forma de uma faixa ao longo da diagonal principal da matriz de distâncias. Assim, são calculados apenas $(k + l + 1) \times min[m, n]$, onde m e n são os comprimentos das seqüências S e T comparadas.

No entanto, este algoritmo só oferece os referidos ganhos na obtenção dos alinhamento globais ótimos quando as seqüências analisadas são razoavelmente semelhantes. Quanto maior a diferença entre elas, maior acabará sendo o valor final que k e l assumirão durante a execução do algoritmo.

As Figura 2.18 ilustra a faixa de valores processados e a relação destes valores

com k e l.

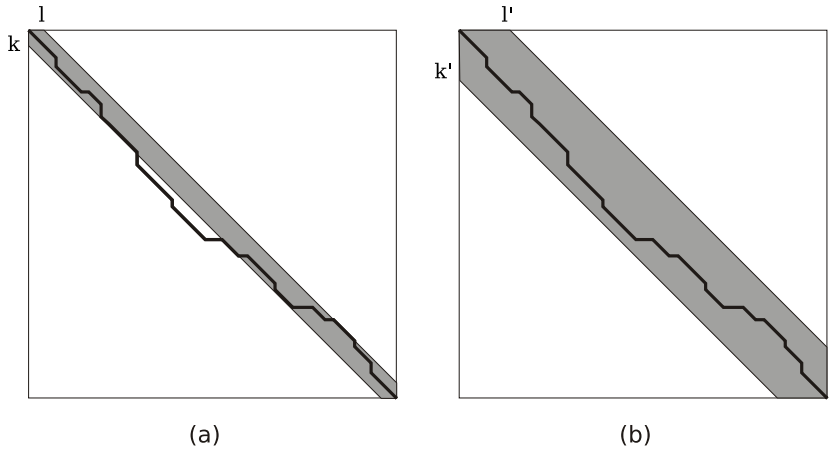


Figura 2.18: Processamento do algoritmo de Fickett. (a) Uma estimativa muito pequena para d leva a valores de k e l que não permitirão a obtenção do alinhamento ótimo. (b) Uma estimativa $d' > d$ maior que o valor de distância do alinhamento ótimo tem como conseqüências k' e l' a partir dos quais é possível recuperar o alinhamento ótimo.

Capítulo 3

Computação em *clusters*

3.1 Visão Geral

3.1.1 *Clusters*

Clusters de computadores fazem parte do conjunto de arquiteturas paralelas. Os sistemas computacionais tradicionais têm seus elementos integrados em um único computador. Os sistemas paralelos e distribuídos diferem dos sistemas tradicionais, pois os seus elementos encontram-se alocados em vários computadores. Estes vários computadores realizam tarefas muitas vezes distintas, mas correlacionadas, comunicando-se entre si para atingir um único objetivo.

A. Vrenios [54] define um *cluster* como um conjunto de computadores que são conectados através de um hardware e software especializados, apresentando uma imagem única de sistema (*single-system image*) aos seus sistemas e usuários. O objetivo da imagem única de sistema é criar a ilusão de que os vários computadores que compõem o *cluster* comportam-se como um único computador.

Uma das classes mais famosas de *clusters* são os da classe Beowulf. A NASA iniciou em 1994 o Projeto Beowulf, criando um supercomputador a partir de 16 computadores pessoais de última geração e mecanismos de comunicação especializados. Havia na época computadores com o mesmo poder computacional do *cluster* inteiro, só que tal equipamento custava cerca de dez vezes mais que o custo total do *cluster* Beowulf.

3.1.2 Características dos *clusters*

3.1.2.1 Alto desempenho

Existem vários motivos que levaram ao estudo dos *clusters*. Dentre estes, podemos citar os primordiais como alto desempenho, escalabilidade e alta disponibilidade. Quando se deseja resolver um problema computacional, uma variável importante a se levar em conta é o tempo necessário para que um computador produza uma solução de uma determinada instância de um problema. É comum nos depararmos com problemas que levam mais tempo que o desejável para serem solucionados computacionalmente. Mesmo um microprocessador de ponta pode não ser capaz de responder no tempo desejado a um determinado problema. No entanto, um

cluster de computadores possui uma relação custo/benefício muito vantajosa. Os processadores cada vez potentes são capazes de, juntos, fornecer alto desempenho computacional a baixo custo.

Pode acontecer, em alguns casos, de o problema computacional exigir a utilização de *clusters*, não por demandar alto desempenho, mas sim por exigir uma quantidade de memória disponível muito maior que a instalável em apenas um computador. O domínio do problema é então dividido para que a massa de dados possa ser tratada por vários computadores.

3.1.2.2 Escalabilidade

Agregar vários computadores para que estes somem seus potenciais computacionais possui um custo. Este custo está relacionado, entre outros fatores, à necessidade que os componentes de um *cluster* têm de trabalharem de forma organizada com um objetivo bem definido.

Adicionar novas máquinas a um *cluster* produz um impacto sobre o desempenho do mesmo. Para que estas trabalhem de forma organizada, devem se utilizar de uma infra-estrutura de comunicação. Quanto maior o número de máquinas utilizando o canal de comunicação, maior a chance de uma máquina encontrar este mesmo canal ocupado. Isto acaba por introduzir atrasos que diminuem a eficiência do sistema como um todo. Em um determinado ponto, adicionar um computador a um *cluster* pode deixar de oferecer ganho de desempenho.

Este ponto é chamdado de ponto de retorno decrescente (*point of diminishing returns*). O custo de gerenciar mais um computador no *cluster* torna-se maior que o poder de processamento do computador agregado ao *cluster*. Este ponto é inevitável e muitos estudos tentam diminuir a velocidade com que se chega a este ponto. Desta forma, quanto mais escalável um *cluster*, mais computadores este comporta até atingir o ponto de retorno decrescente.

3.1.2.3 Alta disponibilidade

Outro motivo que torna os *clusters* atraentes é a *alta disponibilidade* e *tolerância a falhas* que estes são capazes de proporcionar. Alguns serviços necessitam não apenas de bons tempos de resposta, mas também de estarem em funcionamento 24 horas por dia, 7 dias por semana. Se um único computador é responsável por responder a um grande número de requisições, mesmo que este seja capaz de responder a todas estas requisições em tempo hábil, se este computador falhar, nenhuma requisição mais será processada.

Um *cluster* pode ser arquitetado de forma a delegar a cada uma dentro de uma série de máquinas um subconjunto das requisições recebidas. A falha em uma das máquinas certamente significaria mais trabalho às outras máquinas restantes, possivelmente degrandando o tempo de resposta, mas em muitos casos é melhor uma resposta atrasada do que resposta nenhuma.

Alta disponibilidade e tolerância a falhas são conceitos distintos, embora muitas vezes confundidos entre si. A Alta disponibilidade é um qualidade perceptível fora do sistema. Quando o sistema não é projetado para apresentar alta disponibilidade, este tende a falhar mais. Isto é facilmente percebido quando o sistema

não mais responde a requisições externas. Já a tolerância a falhas é mais transparente, significando que o sistema admite falhas, mas absorve até certa extensão os efeitos prejudiciais destas. Um usuário externo não pode determinar, com certeza, até que ponto os mecanismos de tolerância a falhas de um sistema estão em atividade.

3.1.2.4 Sobreposição (*overlap*)

Recentemente, uma das qualidades que mais têm sido exploradas nos computadores é a capacidade de sobreposição (overlap) de tarefas. Sempre que duas ou mais tarefas podem ser realizadas concomitantemente, diz-se que estas podem ser sobrepostas, sendo realizadas em paralelo.

As arquiteturas de computador compartilham, quase sempre, uma estrutura que contém processador, memória e dispositivos de entrada e saída (E/S). Os primeiros computadores não exploravam devidamente a sobreposição de tarefas. Quando uma região de memória era solicitada pelo processador, este ficava aguardando até que os valores correspondentes estivessem disponíveis no barramento. Operações de E/S também mantinham o processador aguardando até que fossem terminadas.

A utilização de técnicas de *pipelining* permitiu que os processadores executassem partes de várias instruções em um mesmo ciclo. Técnicas de predição e *cache* são utilizadas para que, enquanto um processador executa instruções, as próximas posições de memória a serem requisitadas já estejam sendo recuperadas pelo barramento de memória. A utilização de DMA (direct memory access) permite que dispositivos de E/S tenham mais autonomia, transferindo dados diretamente de/para regiões de memória sem onerar o processador.

Muitos problemas solúveis computacionalmente também apresentam alto potencial de sobreposição. Desenvolvendo-se algoritmos que explorem o paralelismo de um problema, pode-se dividir o trabalho necessário para resolvê-lo entre vários computadores. Quando um problema envolve processar um conjunto de dados, em casos mais simples, pode-se dividir estes dados entre vários computadores. Cada um então processa os seus dados e logo em seguida todos os computadores reunem os dados processados, formulando a solução para o problema. Infelizmente, a maioria dos problema não pode ser implementada em um *cluster* de forma simples como a descrita.

3.1.2.5 Lei de Amdahl

Existem limites quanto aos ganhos que podem ser obtidos explorando-se o paralelismo. Gene Amdahl [4] mostrou que há sempre uma porção dos problemas que não podem ser acelerados explorando-se a sobreposição. A Equação 3.1 apresenta uma formalização simples a respeito deste problema.

$$\text{tempo total de execução} = \text{trecho paralelo} + \text{trecho seqüencial} \tag{3.1}$$

Supondo-se que o trecho paralelo do problema possa ser executado por N processadores e que estes processadores consigam dividir o trabalho da melhor

forma possível, temos que o novo tempo total de execução está mostrado na Equação 3.2.

$$\text{tempo total de execução } = \frac{\text{trecho paralelo}}{N} + \text{ trecho seqüencial} \qquad (3.2)$$

A Equação 3.2 é conhecida como Lei de Amdahl. Tai lei mostra que há sempre um limite inferior no que se refere ao tempo de execução possível para uma tarefa. Supondo que o número de processadores N tenda a infinito, e desconsiderando-se o ponto de retorno decrescente, o tempo de execução do trecho paralelo tenderia a zero. O trecho seqüencial ainda assim permaneceria inalterado, o que limita os ganhos possíveis com a exploração do paralelismo.

No mundo real, a situação é ainda pior. O ponto de retorno decrescente limita o número N de processadores que podem ser utilizados. Supondo-se que estes processadores ainda tenham que concorrer uns com os outros pelo uso exclusivo de um determinado recurso, por exemplo E/S, nota-se que o ganho efetivo pode estar longe do ótimo.

Mesmo diante destas dificuldades, a utilização de *clusters* de computadores ainda apresenta um grande atrativo, uma vez que o custo do mesmo é geralmente uma ordem de magnitude menor que o de um supercomputador de poder computacional correspondente.

3.1.3 Taxonomia

Os *clusters* são apenas uma dentre várias arquiteturas que exploram o paralelismo. A seguir serão descritas várias taxonomias criadas para classificar diversas arquiteturas de computação paralela e distribuída.

3.1.3.1 Multiprocessadores e multicomputadores

As arquiteturas tradicionais possuem geralmente um processador, um espaço de memória e dispositivos de E/S. Alguns sistemas possuem vários processadores que compartilham um mesmo barramento, uma mesma memória e os mesmos dispositivos de E/S. Estes são os chamados multiprocessadores.

O projeto de um sistema como este envolve uma série de novos problemas que devem ser observados. Os vários processadores podem executar programas ao mesmo tempo. Isso aumenta o desempenho do sistema, mas aumenta também a concorrência pela memória e por outros recursos. Por estarem compartilhando uma única memória, os processos que rodam ao mesmo tempo agora devem estar atentos ao fato que estes podem operar sobre um mesmo endereço de memória. Isso pode levar a problemas de consistência que serão detalhadosna seção 3.2.

Os multiprocessadores geralmente são arquiteturas de hardware, desenvolvidas para garantir que o desempenho de tal sistema esteja perto do ideal. Isso eleva em muito o custo de tal arquitetura. Exemplos de sistemas multiprocessados são as arquiteturas SMP(*Symmetric Multi-Processor*).

Os multicomputadores, por sua vez, são construídos interligando-se vários computadores. Tais computadores não foram projetados inicialmente para operar em conjunto com outros em paralelo. Uma rede de interconexão rápida é

utilizada para que estes computadores se comuniquem de forma eficiente. Sofwares específicos e bibliotecas de programação são utilizados para criar uma imagem única de sistema. Os *clusters* de computadores são um exemplo de multicomputador.

3.1.3.2 Flynn

Em 1972, M. J. Flynn[16] publicou uma taxonomia que levava em conta qual era o conjunto de instruções sendo executado ao mesmo tempo, bem como analisava quais eram os conjutos de dados sendo processados em um determinado instante. Cada um desses critérios dava origem a duas classificações possíveis: único (single) e múltiplo (multiple). O conjunto de possíveis arquiteturas veio a ser o seguinte:

- SISD - Single Instruction Single Data
- SIMD - Single Instruction Multiple Data
- MISD - Multiple Instruction Single Data
- MIMD - Multiple Instruction Multiple Data

As arquiteturas SISD são representadas pelas tradicionais arquiteturas de Von Neumann, onde um fluxo de instruções opera sobre um fluxo de dados. É uma denominação que não vingou, entrando em desuso.

As arquiteturas SIMD podem ser exemplificadas através dos processadores vetoriais. Um fluxo de instrução é executado. Algumas das instruções executadas são capazes de instruir o processador a realizar uma operação sobre todo um conjunto de dados. Uma instrução de adição pode ser emitida para incrementar em uma unidade o valor de todos os elementos de um vetor. Essa adição será realizada sobre mais de um elemento ao mesmo tempo.

Os *clusters* são um exemplo de arquitetura MIMD, onde cada computador possui um fluxo distinto de instruções sendo executadas em um determinado momento. Além disso, cada computador possui o seu próprio fluxo de dados. O termo MIMD, na verdade, abarca um grande conjunto de arquiteturas paralelas existentes.

Já o termo MISD simplesmente não conseguia descrever com precisão nenhum tipo de arquitetura existente. A idéia de várias operações sendo realizadas sobre um determinado dado não se adequava praticamente a nada. Alguns interpretavam que processadores *pipeline* seriam exemplos de tal categoria, mas por falta de unanimidade, esta denominação também acabou caindo em desuso.

3.1.3.3 Tanenbaum

Como o termo MIMD acaba por denominar uma ampla gama de arquiteturas, A. S. Tanenbaum [50] acabou por criar uma classificação só para esta classe. A Figura 3.1 exibe tal taxonomia.

Primeiramente, os sistemas MIMD foram classificados como fracamente acoplados e fortemente acoplados.

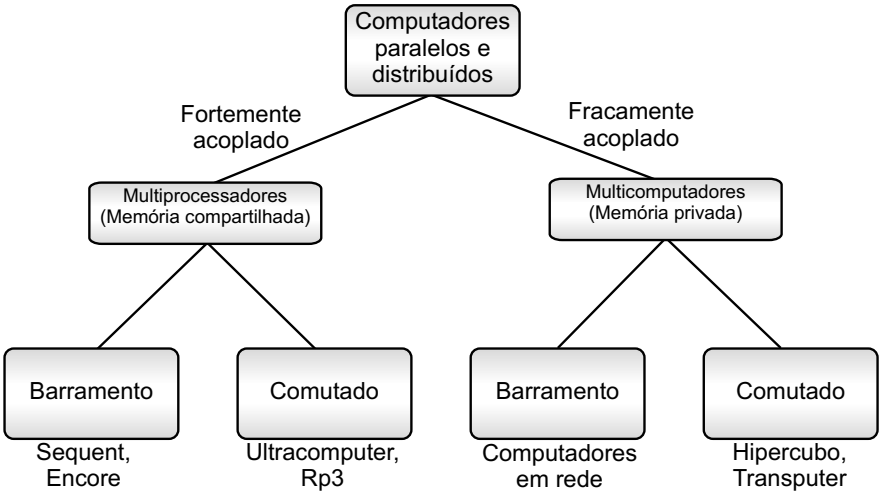


Figura 3.1: Taxonomia de sistemas paralelos e distribuídos de acordo com Tanenbaum [50]

Nos sistemas fracamente acoplados, cada processador possui um espaço privativo de memória. Isto implica que nenhum processador pode, por si só, acessar as posições de memória de outro processador. Para tanto, para que um processador A acesse o conteúdo da memória de um processador B, este último deve ser requisitado a realizar a tarefa e enviar o resultado para A. Este tipo de sistema apresenta a desvantagem de o tempo decorrido para realizar uma operação como a descrita ser ordens de magnitude maior que o tempo de um acesso convencional a um endereço de memória.

Uma das vantagens de um sistema como esse é que, se bem projetado, este pode ser tolerante a falhas. Os *clusters* de computadores são um exemplo de arquitetura fracamente acoplada.

Já em um sistema fortemente acoplado, todos os processadores possuem acesso direto à memória do sistema. Há apenas um espaço de endereçamento. Desta forma, suponhamos que um processador A escreva o valor 10 na posição de memória 500. Se um processador B, logo em seguida, realizar uma leitura da posição 500, este irá encontrar o valor 10 que foi armazenado por A. Diz-se que este sistema possui uma memória compartilhada. A grande vantagem de um sistema com memória compartilhada é a baixíssima latência de comunicação entre processadores. Sistemas SMP são exemplos de arquiteturas fortemente acopladas.

Além disso, essas categorias podem ainda ser subdivididas levando-se em conta a interconexão dos processadores ou computadores. A interconexão pode ser em barramento ou comutada. A interconexão em barramento é provida através de uma rede que está conectada a todos os pontos. Quando dois pontos estão se comunicando, este meio fica ocupado. Enquanto durar a ocupação do meio, este fica impossibilitado de conduzir outra comunicação.

Além disso, o barramento possui uma baixa escalabilidade. À medida que mais processadores utilizam o barramento, maior se torna a concorrência pelo mesmo e, conseqüentemente, maior a latência de comunicação. A vantagem do barramento é a simplicidade de implementação e baixo custo. A Figura 3.2 ilustra

um exemplo de barramento.

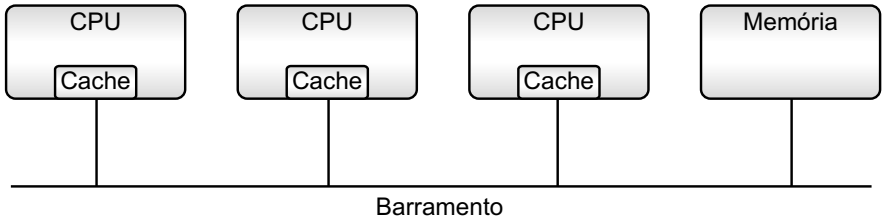


Figura 3.2: Rede de interconexão em barramento

Uma interconexão comutada é capaz de permitir a comunicação entre mais de um par de pontos num mesmo instante. Para tanto, não se pode utilizar apenas um meio de comunicação como acontece no barramento. Um exemplo de rede comutada é a rede *crossbar*. Seja N o número de processadores e M o número de bancos de memória, o switch crossbar possuirá $N \times M$ pontos de interligação. Isso permite que cada banco de memória esteja disponível para um processador diferente num mesmo instante. A Figura 3.3 ilustra um exemplo de interconexão comutada.

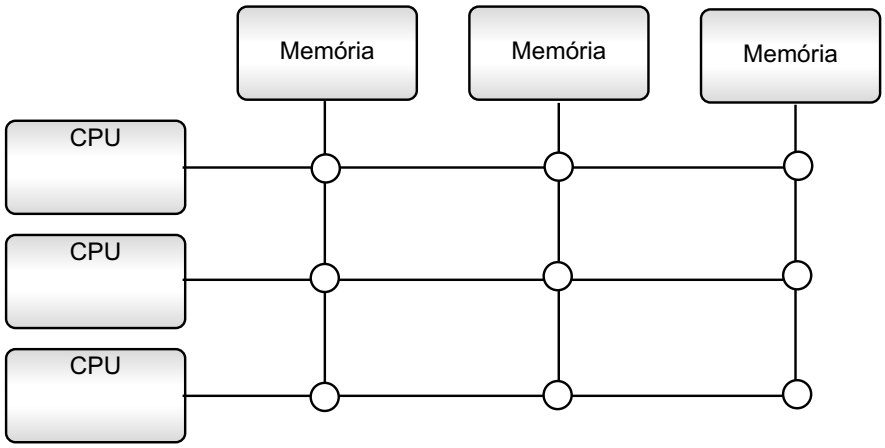


Figura 3.3: Rede de interconexão comutada

3.1.3.4 UMA, NUMA e NORMA

Uma outra classificação possível para os sistemas multiprocessados e multicomputadores é baseada no tempo de acesso aos endereços de memória. Desta forma, podem ser definidas as seguintes categorias:

- Uniform Memory Access (UMA) - Nesta classe de sistemas, todos os processadores possuem a mesma latência de acesso à memória. Todas as operações consomem o mesmo tempo, independente de onde o banco de memória acessado se encontre.

- Non-Uniform Memory Access (NUMA) - Aqui o tempo de acesso à memória pode variar dependendo do endereço solicitado. Uma requisição a um endereço que se localize na mesma máquina que a do processador solicitante será respondida no menor tempo possível. Quando o endereço solicitado encontra-se fisicamente em outra máquina / processador, a rede de intercomunicação é acionada para recuperar ou armazenar o valor solicitado. A própria rede de interconexão é responsável por recuperar no destino os valores solicitados. Não há intervenção do processador da máquina acessada. Neste caso, a latência de acesso à memória não é regular.

- No Remote Memory Access (NORMA) - Algumas arquiteturas, no entanto, não foram projetadas de forma a permitir que a rede de interconexão seja capaz de efetuar operações remotas de memória. Em tais arquiteturas, torna-se imprescindível a cooperação do processador de uma máquina para que a sua memória seja acessada remotamente. É a arquitetura mais lenta de todas pois vale-se, além da rede de interconexão, de software especializado para poder responder à requisições de memória.

3.2 Programação Paralela

3.2.1 Modelos de programação

Até agora foram discutidas diversas arquiteturas de clusters, o que dá uma idéia de como o hardware destes está organizado. No entanto, para que um *cluster* cumpra a sua função, este deve ser programado para realizar uma tarefa específica. Serão abordadas nesta seção uma série de conceitos peculiares à programação em clusters.

Uma das primeiras coisas que devem ser decididas antes de se tentar resolver um problema utilizando-se um *cluster* é definir qual será o modelo de programação utilizado. Existem vários tipos de modelos de programação e alguns são citados aqui:

- Modelo monoprocessador: Também conhecido como modelo Von Neumann. É o modelo mais comunmente utilizado. Na grande maioria das vezes, o programador nem se dá conta de que está utilizando este modelo de programação. Neste modelo, o programador define um conjunto de instruções e uma ordem na qual estas devem ser executadas.

- Modelo multiprocessador simétrico: Conhecido também como modelo de memória compartilhada. Neste modelo, o programador deve estar ciente da existência de vários processadores compartilhando um mesmo espaço de endereçamento de memória. A comunicação entre processadores pode ser feita utilizando-se a memória compartilhada.

- Modelo passagem de mensagens: Neste modelo, o programador deve estar ciente que cada processador possui o seu próprio espaço de endereçamento de memória. A comunicação entres estes processadores pode ser realizada

através de mensagens que são enviadas e recebidas através de uma rede de interconexão.

Algo importante que deve ser notado é que o modelo de programação não está vinculado à arquitetura do sistema. Um programador utiliza-se do modelo Von Neumann para criar um código *assembly* que será executado em um processador, por exemplo, da arquitetura Intel x86. Ocorre, no entanto, que os processadores atuais possuem otimizações arquiteturais, como o *pipelining*, que permitem que estes executem várias instruções ao mesmo tempo e muitas vezes fora da ordem estipulada pelo programa. Apesar disso, o processador possui um comportamento aparente, em contraste com o comportamento interno, onde o resultado final do programa é exatamente aquele esperado pelo programador.

Os modelos de programação em memória compartilhada e com passagem de mensagens são exemplos de modelos de programação paralela. A importância do modelo de programação está no fato que este determina que tipos de decisões e estratégias deverão ser tomadas para se conseguir bons resultados quando se tenta explorar o paralelismo dos problemas.

3.2.2 Projeto de algoritmos paralelos

Foster [18] descreve uma metodologia que pode ser utilizada para se tentar explorar o paralelismo dos problemas. O método consiste em quatro passos que podem ser definidos como particionamento, comunicação, aglomeração e mapeamento. Apesar de existir uma certa seqüência na execução destes passos, decisões tomadas em um passo podem demandar a revisão de decisões tomadas anteriormente.

- Particionamento: A idéia por trás do particionamento é encontrar o potencial paralelismo de um problema. Isso implica em dividir em pedaços a computação e os dados associados ao problema. Pode-se abordar inicialmente tanto um quanto o outro. Primar pela divisão da computação é chamado decomposição funcional do problema, enquanto que primar pela divisão dos dados chama-se decomposição de domínio do problema. Um bom particionamento é imprescindível para que se possa expor ao máximo o paralelismo de um problema.

- Comunicação: As tarefas definidas durante o particionamento são, em maior ou menor grau, dependentes entre si. Esta dependência força estas tarefas a se comunicarem, pois alguns dados necessários por uma tarefa estão muitas vezes em posse de outra tarefa. Existem várias formas pelas quais estas tarefas podem se comunicar. Há as comunicações locais, ou ponto-a-ponto, onde uma tarefa comunica-se com uma pequena quantidade de outras tarefas. Já as comunicações globais envolvem um grande grupo de tarefas ou até mesmo todas as tarefas envolvidas em uma solução. Uma boa escolha das estruturas e métodos de comunicação é importante para minimizar o custo de comunicação entre as tarefas.

- Aglomeração: A aglomeração tem por finalidade reunir as várias tarefas definidas no passo de particionamento de forma a criar tarefas maiores. Um

dos objetivos da aglomeração é diminuir o número de tarefas, conseqüentemente diminuindo o custo de comunicação entre elas. Um número muito grande de tarefas acaba por exigir muita comunicação, enquanto que um baixo número de tarefas acaba por limitar a exploração do paralelismo do problema. O objetivo é encontrar o equilíbrio na relação tarefas $\times$ comunicação.

- Mapeamento: Definidos os grupos de tarefas, estas devem então ser mapeados para os processadores que efetivamente irão executá-las. Deve-se estar atento ao fato que algumas tarefas definidas no passo anterior podem demandar maior poder computacional que outras. Isso acabaria diminuindo o paralelismo efetivamente explorado pela solução. Boas soluções de balanceamento de carga devem ser vislumbradas para que o poder computacional disponível seja distribuído homogeneamente sobre as tarefas a serem executadas.

Capítulo 4

Soluções paralelas e distribuídas

A utilização de técnicas de programação paralela e distribuída na bioinformática é fruto inevitável de uma combinação de múltiplos fatores. O primeiro fator a contribuir para tal são os algoritmos utilizados na obtenção dos alinhamentos.

Os algoritmos vistos no Capítulo 2 possuem complexidade de tempo $O(n^2)$ em função do tamanho da entrada. Neste caso, as entradas são as seqüências genéticas cujos alinhamentos se deseja obter. Para se ter uma idéia da dimensão do problema, a Tabela 4.1 exibe o comprimento aproximado, em número de pares de bases nitrogenadas que cada organismo exemplificado possui.

Comprimento	Organismo
50.000	Mitocondrial
500.000	Saccharomyces cerevisiae (Fermento)
5.000.000	Bactéria
50.000.000	Cromossomo Humano - curto
500.000.000	Cromossomo Humano - extenso
5.000.000.000	Genoma completo de um mamífero

Tabela 4.1: Exemplos de comprimentos, em pares de bases, de algumas seqüências de DNA encontradas na natureza[37]

Atualmente, as comparações entre seqüências da ordem dezenas de milhões de pares de bases já têm provado ser desafiantes. Tais comparações levariam, hoje em dia, algumas semanas para produzir o alinhamento entre apenas duas seqüências[37].

Tal demanda de tempo por si só já se apresenta como uma grande restrição nas pesquisas da área. Deve-se juntar a isso o fato de que, a cada dia, mais e mais organismos têm os seus genomas seqüenciados. Uma vez que cada genoma seqüenciado deve então ser comparado com os outros genomas já conhecidos, a cada dia que passa, o número de possíveis comparações entre genomas cresce vertiginosamente.

Toda vez que novas seqüências de interesse, juntamente com os seus genes e proteínas relacionados, são descobertos, estes se tornam patrimônio intelectual do pesquisador ou laboratório responsável pela pesquisa. A corrida pelas patentes

sobre os genes descobertos vem aumentar ainda mais a necessidade por algoritmos e soluções cada vez mais eficientes para a produção de alinhamentos.

O problema do tempo necessário para se alinhar seqüências tem sido tratado através de simplificações do problema. A utilização de algoritmos como o BLAST[2], o FASTA[43] e o recente DASH[20] torna possível a obtenção de resultados em tempo mais hábil. Tais algoritmos, apesar de mais rápidos, produzem resultados menos precisos em decorrência das técnicas heurísticas utilizadas nos mesmos. Tal precisão é referida como *sensibilidade*[47], que é diferença entre o número de alinhamentos produzidos pela heurística e o número de alinhamentos produzidos por uma execução de um algoritmo ótimo.

É muito comum soluções paralelas e distribuídas se valerem de alguma forma de heurística para obter bons desempenhos. Mesmo quando as heurísticas promovem altos níveis de sensibilidade, resta a dúvida se os alinhamentos não encontrados são aqueles menos significativos, principalmente quando se trata do ponto de vista biológico. Desta forma, o balanceamento entre desempenho e sensibilidade é vital para o problema em questão.

Quando se parte do algoritmo de Smith-Waterman para produzir soluções paralelas para os problemas em questão, esbarra-se numa dificuldade introduzida pela própria técnica de programação dinâmica. A produção de cada elemento $S_{i,j}$ da matriz de similaridades depende dos valores $S_{i-1,j}$, $S_{i-1,j-1}$ e $S_{i,j-1}$ da mesma matriz. A grande maioria das soluções paralelas para o problema têm de enfrentar tal restrição, apresentando, em maior ou menor grau, alguma forma de processamento em onda(*Wavefront*)[45].

O método de processamento em onda apresenta níveis não uniformes de paralelismo. No início do processamento, apenas um elemento da matriz de similaridades precisa ser calculado. Não há paralelismo nenhum neste ponto. À medida que os valores da matriz vão sendo calculados, ao longo das antidiagonais da matriz, o nível de paralelismo vai aumentando. A partir de um certo ponto, o nível de paralelismo volta a decair. Isso limita, de alguma forma, o ganho de desempenho que é possível obter com tal método. A Figura 4.1 ilustra esse conceito.

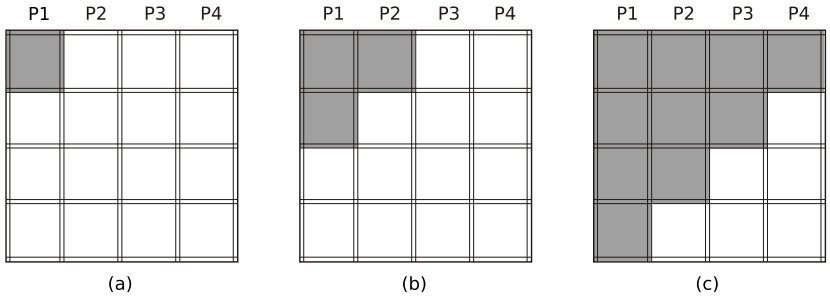


Figura 4.1: Processamento em onda[45] (*wavefront*) para 4 processadores. (a) No início do processamento, apenas P_1 tem sua dependência de dados satisfeita. (b) A frente de onda avança, aumentando o número de processadores sendo utilizados em paralelo. (c) A anti-diagonal onde todos os processadores estão ocupados. A partir deste ponto a distribuição de trabalho volta a se tornar cada vez mais desigual.

Desta forma, para que uma solução paralela para este problema seja capaz

de obter um bom desempenho, é desejável que o problema do processamento em onda seja trabalhado. Algumas soluções tentam minimizar o impacto resultante deste problema, enquanto outras removem tal dependência entre os nós ao custo de alguma sensibilidade.

Outro aspecto que está intimamente ligado à dependência de dados é a comunicação do mesmos entre os processadores, através da rede de interconexão. A utilização de uma infraestrutura de computação pública [44] ou de computação em grade [1] mostram que a infraestrutura tem papel de suma importância sobre o desempenho da solução. Para que estas infraestruturas possam vir a produzir desempenhos melhores que os que têm sido produzido, é necessário antes de mais nada prover uma solução adequada para a dependência de dados.

Diversas técnicas já foram desenvolvidas e testadas para lidar com os problemas descritos até este ponto. Tais técnicas serão comentadas a seguir.

4.1 Prefixo paralelo

Aluru *et. al.*[3] descrevem em seu artigo um meio de se particionar a matriz de similaridades de forma a produzir uma distribuição uniforme de trabalho entre cada processador. Tal distribuição é obtida através do uso de uma técnica denominada *prefixo paralelo*, a fim de se evitar um processamento em ondas.

O prefixo paralelo é definido da seguinte forma. Considere n itens de dados $x_0, x_1, \ldots, x_{n-1}$ e um operador binário associativo $\otimes$ que opera sobre os itens de dados e produz um resultado do mesmo tipo. Deseja-se computar as n somas parciais $s_0, s_1, \ldots, s_{n-1}$, onde:

$$s_i = x_0 \otimes x_1 \otimes x_2 \otimes \ldots \otimes x_i \tag{4.1}$$

Tal problema pode ser resolvido em tempo $O\left(\frac{n}{p} + (\tau + \mu)\log p\right)$, onde p é o número de processadores, τ é o custo inicial de uma operação de comunicação e μ é o custo de comunicação associado ao tamanho da mensagem.

Enquanto os algoritmos paralelos tradicionais trabalham sobre a antidiagonal da matriz de similaridades, o algoritmo em questão preenche a matriz de similaridades linha por linha, levando a uma distribuição uniforme do trabalho entre os processadores durante todo o processo. Tomando-se a equação de recorrência[3]:

$$\mathrm{S}_{i,j} = max \begin{cases} \mathrm{S}_{i-1,j} - g \\ \mathrm{S}_{i,j-1} - g \\ S_{i-1,j-1} + p(i,j) \end{cases} \tag{4.2}$$

Separando-se os termos precomputados como w_j:

$$w_j = max \begin{cases} \mathrm{S}_{i-1,j} - g \\ \mathrm{S}_{i-1,j-1} + p(i,j) \end{cases} \tag{4.3}$$

$$\text{Então, } \mathrm{S}_{i,j} = max \begin{cases} w_j \\ \mathrm{S}_{i,j-1} - g \end{cases} \tag{4.4}$$

$$
\begin{aligned}
\text{Seja } x_j &= \mathrm{S}_{i,j} - \sum_{k=1}^{j}(-g) \\
&= max \begin{cases} w_j - \sum_{k=1}^{j}(-g) \\ \mathrm{S}_{i,j-1} - g - \sum_{k=1}^{j-1}(-g) \end{cases} \\
&= max \begin{cases} w_j - \sum_{k=1}^{j}(-g) \\ x_{j-1} \end{cases}
\end{aligned}
$$

$$
\text{Seja } y_j = \sum_{k=1}^{j}(-g) \tag{4.5}
$$

$$
z_j = w_j - y_j \tag{4.6}
$$

As n somas parciais $\sum_{k=0}^{j}(-g)$, para $1 \leq j \leq n$ podem ser computadas utilizando-se prefixo paralelo. Como resultado disso, os valores z_j são conhecidos. Então,

$$
x_j = max \begin{cases} z_j \\ x_{j-1} \end{cases} \tag{4.7}
$$

Uma vez que os valores z_j são conhecidos, os valores x_j podem ser computados através de prefixo paralelo utilizando-se max como operador binário associativo.

Assim, $\mathrm{S}_{i,j}$, para $1 \leq i \leq m$ e $1 \leq j \leq n$ é obtido utilizando-se:

$$
\begin{aligned}
\mathrm{S}_{i,j} &= x_j + \sum_{k=1}^{j}(-g) \\
&= x_j + y_j
\end{aligned} \tag{4.8}
$$

As equações 4.3, 4.5, 4.6, 4.7 e 4.8 são usadas para computar a linha da matriz de similaridades usando informação da linha anterior. Calcular as equações 4.5 e 4.7 requer o uso de prefixo paralelo.

Para o caso das funções de *gap* constantes, onde $g \geq 0$, temos:

$$
x_j = max \begin{cases} w_j - gj \\ x_{j-1} \end{cases} \tag{4.9}
$$

$$
\mathrm{S}_{i,j} = x_j - gj \tag{4.10}
$$

Assim, cada linha da matriz pode ser computada usando-se as equações 4.3, 4.9, 4.10, utilizando-se prefixo paralelo para calcular apenas a equação 4.9.

Desta forma obtém-se a matriz de similaridades para o alinhamento global entre S e T. O autor mostra no artigo que esta fase do algoritmo leva um tempo $O(\frac{mn}{p} + \tau(m+p)\log p + \mu m \log p)$, onde cada processador utiliza um espaço $O(\frac{mn}{p})$. Neste ponto, o alinhamento propriamente dito é obtido a partir dos dados

armazenados em cada processador. Essa obtenção leva um tempo $O(m+n)$, que não prejudica o resultado desde que $m+n = O(\frac{mn}{p})$.

O algoritmo também é capaz de obter alinhamentos globais com funções lineares de *gap*. Em vez de utilizar apenas uma matriz S, faz-se necessário utilizar-se três matrizes S_1, S_2 e S_3, todas elas de tamanho $(m+1) \times (n+1)$.

Utilizando-se técnicas descritas por Myers[38], o autor propõe também uma versão mais eficiente do algoritmo no que diz respeito ao espaço utilizado. Obtem-se assim um limite de $O(m + \frac{n}{p})$ para o espaço. No entanto, tal adaptação inviabiliza a utilização de funções lineares de *gap*.

A obtenção de alinhamentos locais também pode ser obtida através do uso de prefixo paralelo. Originalmente a equação de recorrência para $S_{i,j}$ seria:

$$S_{i,j} = max \begin{cases} S_{i-1,j} - g \\ S_{i,j-1} - g \\ S_{i-1,j-1} + p(i,j) \\ 0 \end{cases} \tag{4.11}$$

Alterando-se o termo pre-computado w_j:

$$w_j = max \begin{cases} S_{i-1,j} - g \\ S_{i-1,j-1} + p(i,j) \\ 0 \end{cases} \tag{4.12}$$

Os valores de 4.12 são obtidos utilizando-se prefixo paralelo. Também para o caso de alinhamentos locais o autor mostra como adaptar o algoritmo para funções lineares de *gap*.

Para alinhamentos locais, o algoritmo funciona em três fases. Primeiramente é executado o algoritmo para alinhamentos locais sobre as seqüências $S = s_1, s_2, \ldots, s_m$ e $T = t_1, t_2, \ldots, t_n$. Obtém-se o ponto (i, j), onde está localizado o valor máximo max indicando o final do alinhamento local ótimo. No segundo passo, realiza-se um alinhamento global entre as seqüências invertidas $s_i, s_{i-1}, \ldots, s_1$ e $t_j, t_{j-1}, \ldots, t_1$. Obtém-se desta forma, o ponto (k, l), também com valor max, correspondendo ao ponto inicial do alinhamento local ótimo. O terceiro passo consiste em realizar um alinhamento global entre $s_k, s_{k+1}, \ldots, s_i$ e $t_l, t_{l+1}, \ldots, t_j$. O alinhamento só precisa ser obtido realmente na terceira fase.

Também para este caso, o tempo de execução do algoritmo é da ordem $O(\frac{mn}{p})$.

4.2 Modelo EARTH

Martins *et. al.*[33, 34, 12] abordaram o problema de obter alinhamentos (globais e locais) com uma solução implementada em um *cluster* Beowulf, utilizando um modelo de programação EARTH que se baseia em *threads* de granularidade fina.

O modelo EARTH (*Efficient Architecture for Running THreads*)[26, 52] de execução consiste em uma coleção de *threads* cuja ordem de execução é determinada pelas dependências de dados e controles explicitamente definidos no programa. As *threads*, por sua vez, são divididas em *fibras* que não são preemptivas

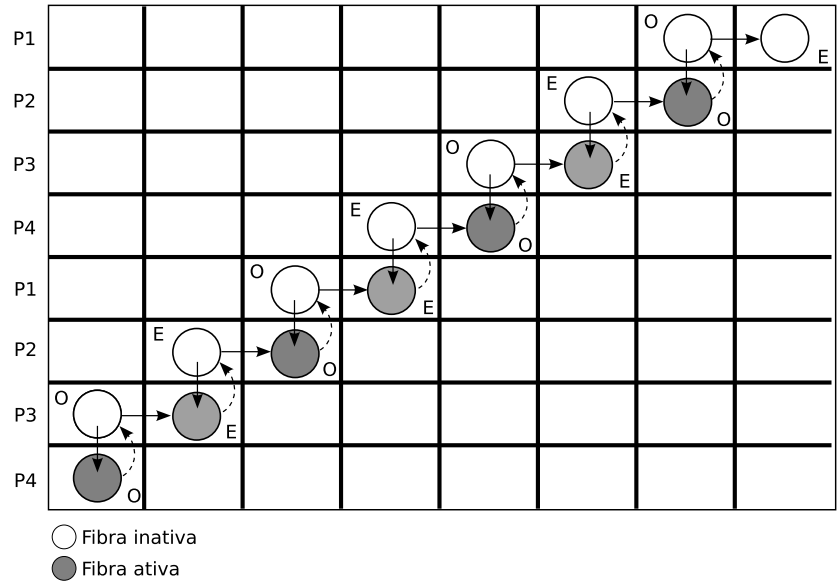


Figura 4.2: Processamento da matriz de similaridades utilizando EARTH[34], [33].

e são disparadas através de mecanismos semelhantes aos encontrados em um *workflow*. O modelo EARTH permite obter vantagens através do uso de vários mecanismos de sincronização e comunicação entre as *fibras* dentro da mesma *thread*, explorando assim a localidade de dados. Permite ainda a sobreposição da computação e da comunicação através de um conjunto de *fibras* prontas para execução que o processador pode disparar assim que outras *fibras* terminem. Sistemas EARTH já foram implementados em várias plataformas: MANNA e PowerMANNA, IBM SP2, Sun SMP Cluster e Beowulf. Programas EARTH são desenvolvidos utilizando-se a linguagem Threaded-C[26, 52], na qual o paralelismo pode ser explicitamente declarado pelo programador.

Em sua solução paralela para o alinhamento local, a matriz de similaridades é dividida em faixas, sendo cada uma destas faixas dividida em blocos retangulares. Uma *thread* é destacada para cada faixa horizontal e o processamento é realizado pelas *fibras* E (ímpares) e O (pares). Enquanto uma *fibra* está processando um bloco, a outra *fibra* está aguardando que as dependências de dados se satisfaçam para iniciar o processamento. O processamento é realizado em onda, como ilustra a Figura 4.2.

O algoritmo registra apenas alinhamentos que possuam valor acima de um limiar definido. Isso porque a quantidade de possíveis alinhamentos encontrados cresce exponencialmente em função do tamanho das seqüências analisadas, como descreve Waterman[55]. Além disso, os alinhamentos são registrados armazenando-se as posições iniciais e finais dos mesmos. Tais valores são obtidos à medida que o algoritmo preenche a matriz de similaridades. Desta forma, o algoritmo economiza espaço, uma vez que não armazena toda a matriz de similaridades. Foram realizados testes de escalabilidade do algoritmo, tendo sido alinhadas com sucesso seqüências de 30K a 900K nucleotídeos. Para as seqüência de 900K nucleotídeos, os speedups absolutos obtidos foi de aproximadamente 40,7 para 64

nós de processamento[34]. O cluster Beowulf utilizado era composto de 64 nodos, cada um com 2 processadores Pentium Pro 200MHz e 128 MB de memória, interligados por um *switch* 100Mbps.

O autor faz comparações para estimar a sensibilidade do algoritmo em relação ao MUMmer (www.tigr.org/tigr-scripts/ CMR2/webmum/mumplot) e ao BLAST (www.ncbi.nlm.nih.gov/blast/bl2seq/ bl2.html). São comparados os genomas mitocondriais humano e de camundongo, genomas mitocondriais humano e da mosca drosófila e genomas de *Mycoplasma pneumoniae* e de *Mycoplasma genitalium*. Os testes realizados mostraram que a solução em questão foi capaz obter uma sensibilidade bem superior à do MUMmer e à do BLAST.

4.3 PSW-DC (*Divide and Conquer*)

Zhang *et. al.*[58] descrevem um algoritmo Smith-Waterman paralelo, denominado PSW-DC, que utiliza a técnica *dividir e conquistar*. O algoritmo aborda especificamente o problema de obter o melhor alinhamento local entre duas seqüências, sendo utilizada uma função constante de *gaps*.

A solução apresenta um paralelismo de granularidade esparsa, devido à técnica dividir-e-conquistar. Consiste basicamente de três fases: Na primeira, os dados são particionados de acordo com o número de processadores e distribuídos entre os mesmos. Na segunda fase, o algoritmo Smith-Waterman é executado de forma independente em cada processador. Na terceira fase, os resultados intermediários obtidos em cada processador são combinados e o melhor alinhamento local é produzido.

Sendo assim, tomando-se S e T como as seqüências analisadas e p como o número de processadores em paralelo, a seqüência S é dividida em p subseqüências, designadas $S_0, S_1, \ldots, S_{p-1}$ onde o comprimento de cada uma dessas subseqüências mede $\frac{|S|}{p}$. Cada seqüência S_i é comunicada ao processador P_i e a seqüência T é transmitida para todos os processadores.

Cada processador produz uma matriz de similaridades D_i a partir da execução do algoritmo de Smith-Waterman utilizando as seqüências S_i e T, produzindo matrizes de tamanho $\left(\frac{|S|}{p} + 1\right) \times (|T| + 1)$. Sendo assim, cada processador armazena $\frac{1}{p}$ da matriz de similaridades. A partir de cada matriz D_i, são obtidos o melhor alinhamento $\mathcal{A}_i$ e o seu valor M_i correspondente. No entanto, os alinhamentos $\mathcal{A}_i$ são resultados intermediários para se obter $\mathcal{A}$, que seria o alinhamento ótimo entre S e T.

Para obter $\mathcal{A}$ a partir dos $\mathcal{A}_i$, foi desenvolvida a técnica heurística C&E (combinar e estender). Tal técnica serve para obter os alinhamentos que ultrapassam as fronteiras entre as seqüências S_n e S_{n+1}. A Figura 4.3 mostra as 6 possibilidades que surgem quando são tomadas duas matrizes D_n e D_{n+1}, onde o ponto (i_0, j_0) é o ponto final do alinhamento $\mathcal{A}_0$ e o ponto (i_1, j_1) é o ponto inicial do alinhamento $\mathcal{A}_1$.

Na Figura 4.3a, $j_0 = |T|$ e $j_1 = 1$. Ou seja, o final de $\mathcal{A}_0$ termina no final de T e o início de $\mathcal{A}_1$ começa no início de T. Assim, $\mathcal{A}_0$ não pode ser estendido para baixo e $\mathcal{A}_1$ não pode ser estendido para cima.

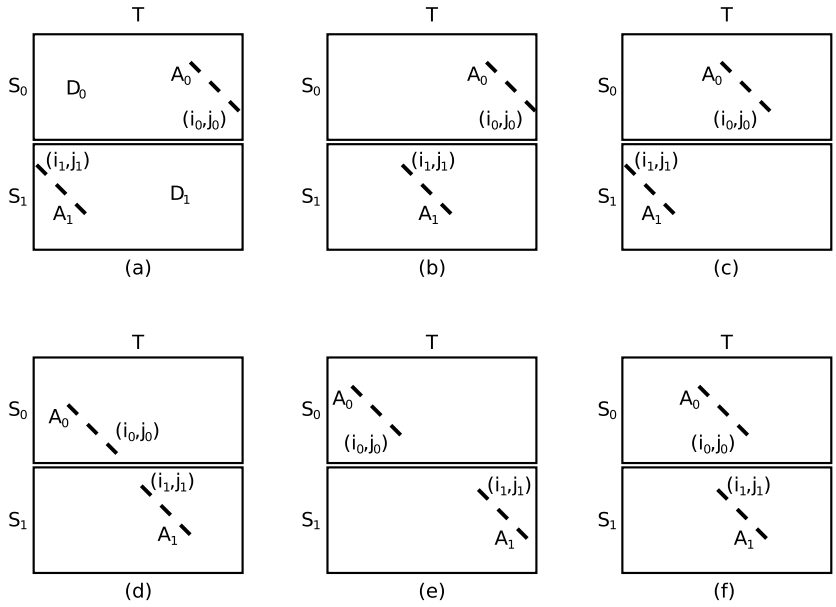


Figura 4.3: Relações possíveis entre dois alinhamentos no PSW-DC.

Na Figura 4.3b, $j_1 > 1$, sendo assim, $\mathcal{A}_1$ pode ser estendido para cima em direção a D_0. Caso a estensão encontre a fronteira entre as matrizes D_0 e D_1, uma mensagem é enviada ao processador que possui D_0, e este continua efetuando a extensão. O caso da Figura 4.3c é análogo ao da Figura 4.3b, só que neste caso só $\mathcal{A}_0$ pode ser estendido para baixo.

Já nas Figuras 4.3d e 4.3e, é preciso definir um valor K, tal que $K = max\left[(|S_0| - i_0 + i_1), (j_1 - j_0)\right]$ e *miscost* sendo a penalidade atribuída aos *gaps*. Na Figura 4.3d $j_0 < |T|, j_1 > 1$ e $j_0 < j_1$ e $K \times miscost < min(M_0, M_1)$. Isto significa que é possível estender $\mathcal{A}_0$ e $\mathcal{A}_1$ de forma que estes componham um só alinhamento sem precisar inserir muitos *gaps*. Na Figura 4.3e, a distância entre $\mathcal{A}_0$ e $\mathcal{A}_1$ é grande o suficiente para que, caso se tente unir os dois alinhamentos, a quantidade de *gaps* produzirá um alinhamento de valor pior do que os já foram encontrados. Assim, $\mathcal{A}_0$ e $\mathcal{A}_1$ são estendidos independentemente.

No caso da Figura 4.3f, ocorrem uma sobreposição entre $\mathcal{A}_0$ e $\mathcal{A}_1$. A extensão procede de forma semalhante ao caso mostrado na figura Figura 4.3e.

Os casos descritos resolvem o problema para $p = 2$. Quando $p > 2$, o método C&E é utilizado da seguinte forma:

1. Ordenar os alinhamentos $\mathcal{A}_i$ decrescentemente, para $i = 0, 1, \ldots, p-1$, de acordo com os seus valores;

2. Sejam $\mathcal{A}_\alpha$ e $\mathcal{A}_\beta$, localizados nos processadores P_α e P_β, os dois melhores alinhamentos;

3. Se $\mathcal{A}_\alpha$ é adjunto a $\mathcal{A}_\beta$, aplica-se o método C&E mostrado na Figura 4.3 para manipular $\mathcal{A}_\alpha$ e $\mathcal{A}_\beta$. O pior alinhamento resultante é removido e o melhor alinhamento resultante é denominado $\mathcal{A}_\alpha$;

4. Se $\mathcal{A}_\alpha$ não é adjunto a $\mathcal{A}_\beta$, aplica-se o método C&E mostrado na Figura 4.3 para manipular $\mathcal{A}_\alpha$ e o seu vizinho que se encontra na direção de $\mathcal{A}_\beta$. O

vizinho de $\mathcal{A}_\alpha$ é removido da lista e o alinhamento resultante é denominado $\mathcal{A}_\alpha$;

4.1 Se $\mathcal{A}_\alpha$ é adjunto a $\mathcal{A}_\beta$ dessa vez, repetir o passo 3;

4.2 Se $\mathcal{A}_\alpha$ não é adjunto a $\mathcal{A}_\beta$ dessa vez, pegar os dois maiores alinhamentos na ordem decrescente e denominá-los $\mathcal{A}_\alpha$ e $\mathcal{A}_\beta$, e então repetir o passo 3;

5. Repetir os passo 3 e 4 até que cada alinhamento $\mathcal{A}_i$ tenha sido manipulado. O $\mathcal{A}_i$ resultante, cujo valor for máximo, será o resultado final.

Apesar de o método C&E permitir a obtenção do alinhamento local em tempo $O\left(\frac{mn}{p}\right)$, este prejudica a precisão do resultado. Os resultados obtidos indicam que, quanto maior o número de processadores utilizados, menor a sensibilidade do algoritmo. Mostram também que quanto maior o comprimento das seqüências, maior a sensibilidade do algoritmo.

O algoritmo foi implementado em um *cluster* chamado DAWNING 2000-I, sendo cada nó composto por um Power PC 604 Dual, de 200MHz, com 256 MB de RAM e disco rígido de 2 GB. A implementação foi feita em linguagem C utilizando comunicação por passagem de mensagens (MPI). Os *speedups* apresentados são bem consistentes, para seqüências de 4K a 16K nucleotídeos, girando entre 9 e 10 para 16 processadores.

4.4 Block-Cyclic *wavefront*

Liu[32] estuda o processamento paralelo em onda à luz dos *design patterns*. Os *design patterns* têm sua origem na programação orientada a objetos, onde classes de problemas semelhantes podem ser resolvidos a partir de abordagens testadas e reutilizáveis. Estas abordagens permitem definir um arcabouço sobre os quais aplicações podem ser desenvolvidas semi-automaticamente, separando-se a estrutura da comunicação do programa paralelo do trecho de implementação seqüencial.

O autor propõe um novo padrão denominado *block-cyclic based wavefront*. Para sistemas paralelos de granularidade esparsa, o processamento em onda deve ser realizado sobre agrupamentos de células, a fim de diminuir a quantidade de comunicação entre os processadores. Figura 4.4a mostra a divisão de uma matriz 8×8 entre 4 processadores e a Figura 4.4b mostra o processamento com blocos 2×2.

O processamento em onda apresenta o inconveniente de não explorar devidamente o poder de processamento disponível. A Figura 4.5 ilustra o método *block-cyclic*. Um fator de divisão é introduzido, induzindo a uma distribuição cíclica das colunas. Tal refinamento permite ocupar mais cedo o poder de processamento disponível, influenciando sensivelmente no desempenho da solução. No entanto, a quantidade de informação trocada entre os processadores aumenta proporcionalmente ao fator de divisão, devendo ser encontrado um equilíbrio entre o fator de divisão e o desempenho da aplicação.

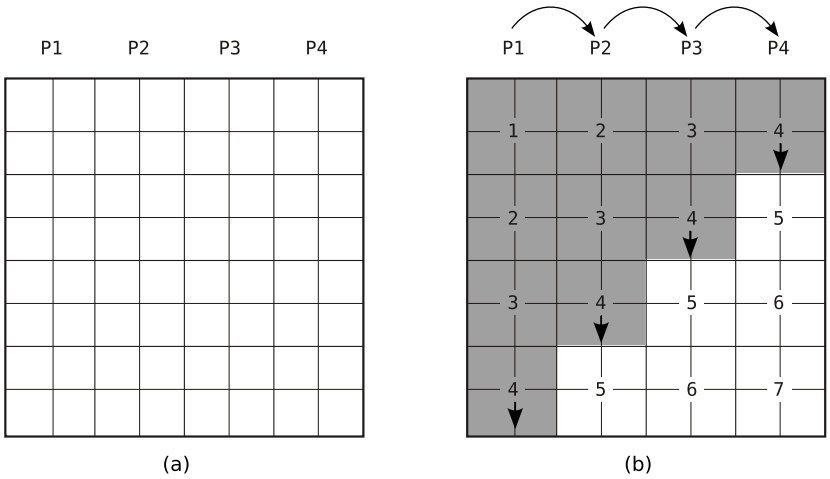


Figura 4.4: Divisão em colunas para $S_{i,j}$. (a) Divisão de colunas para 4 processadores; (b) Processamento em onda para blocos 2×2 [32]

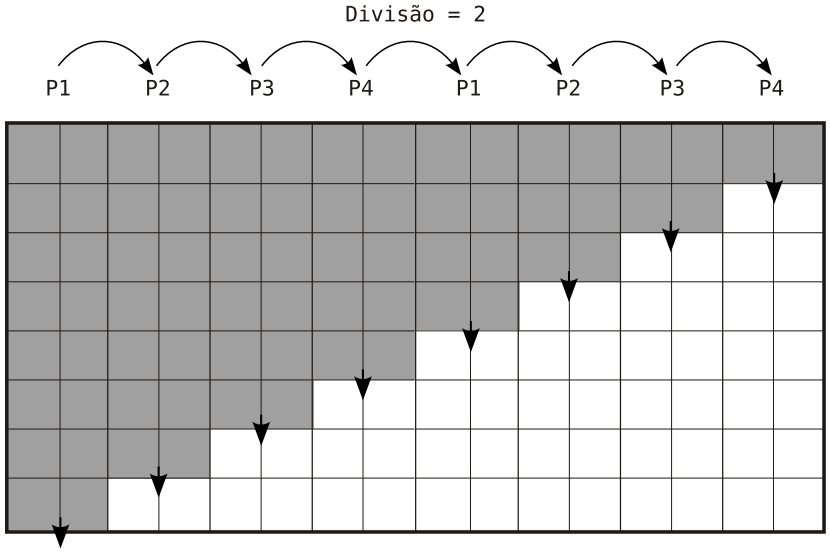


Figura 4.5: Método *block-cyclic* para processamento em onda[32]

Vários algoritmos foram implementados para testar a idéia, sendo estes semelhantes quanto à dependência de dados e demandando alguma forma de processamento em onda. Uma das implementações apresentadas é a do algoritmo Smith-Waterman. A linguagem utilizada foi C++ com MPI. Os testes feitos com o *Block-Cyclic Wavefront* utilizaram seqüências de 100 Kbp. Foi utilizado um *cluster* composto de 16 processadores Intel Pentium IV Xeon 2.6GHz com 512 MB de RAM cada, interconectados por um *switch* Gbit/sec Myrinet.

4.5 FastLSA

Driga *et. al.*[14] produziram uma implementação paralela para o FastLSA[9], um algoritmo de alinhamento global cuja característica principal é a capacidade de adaptar-se à quantidade de memória disponível.

Dada uma função de *gap*, o algoritmo FastLSA produz o mesmo alinhamento ótimo produzido pelos algoritmos de Needleman-Wunsch e Hirschberg. A idéia básica do algoritmo FastLSA é usar mais memória disponível para reduzir a quantidade de processamento realizado pelo algoritmo de Hirschberg. Isto é conseguido através de três adaptações:

1. Dividir ambas as seqüências a serem alinhadas, em vez de dividir apenas uma delas;
2. Dividir cada seqüência em k partes, em vez de dividir apenas em duas partes;
3. Armazenar algumas linhas e colunas específicas da matriz de similaridades no *grid cache* a fim de diminuir o reprocessamentos.

Sejam S e T, onde $m = |S|$ e $n = |T|$, as seqüências as serem alinhadas. Seja RM a quantidade de memória disponível para se executar o algoritmo. Esta quantidade de memória pode representar tanto a memória principal quanto a memória *cache* do processador, dependendo apenas do nível de otimização a que se deseja chegar. Se $RM > m \times n$, então há memória suficiente para se executar, por exemplo, o algoritmo de Needleman-Wunsch, armazenando-se toda a sua matriz de similaridades e obtendo-se o alinhamento (Figuras 4.6a e 4.6b). Esta solução é denominada *Base Case*.

O algoritmo FastLSA resolve o problema de alinhar seqüências utilizando um processamento recursivo. Sendo assim, caso o problema demande uma matriz de similaridades maior que um valor BM estipulado pelo usuário, o problema é subdividido utilizando-se o parâmetro k. São alocadas k linhas e k colunas que irão compor o *grid cache*. A matriz de similaridades é processada em blocos denominados *Fill Cache*, sendo que apenas os valores do *grid cache* são guardados. Ao final, é realizada uma recursão do algoritmo para o *Fill Cache* do canto inferior direito da matriz.

Quando é realizada a recursão, o processo de obtenção do alinhamento envolve concatenar os vários sub-alinhamentos obtidos nas recursões (Figuras 4.6c até

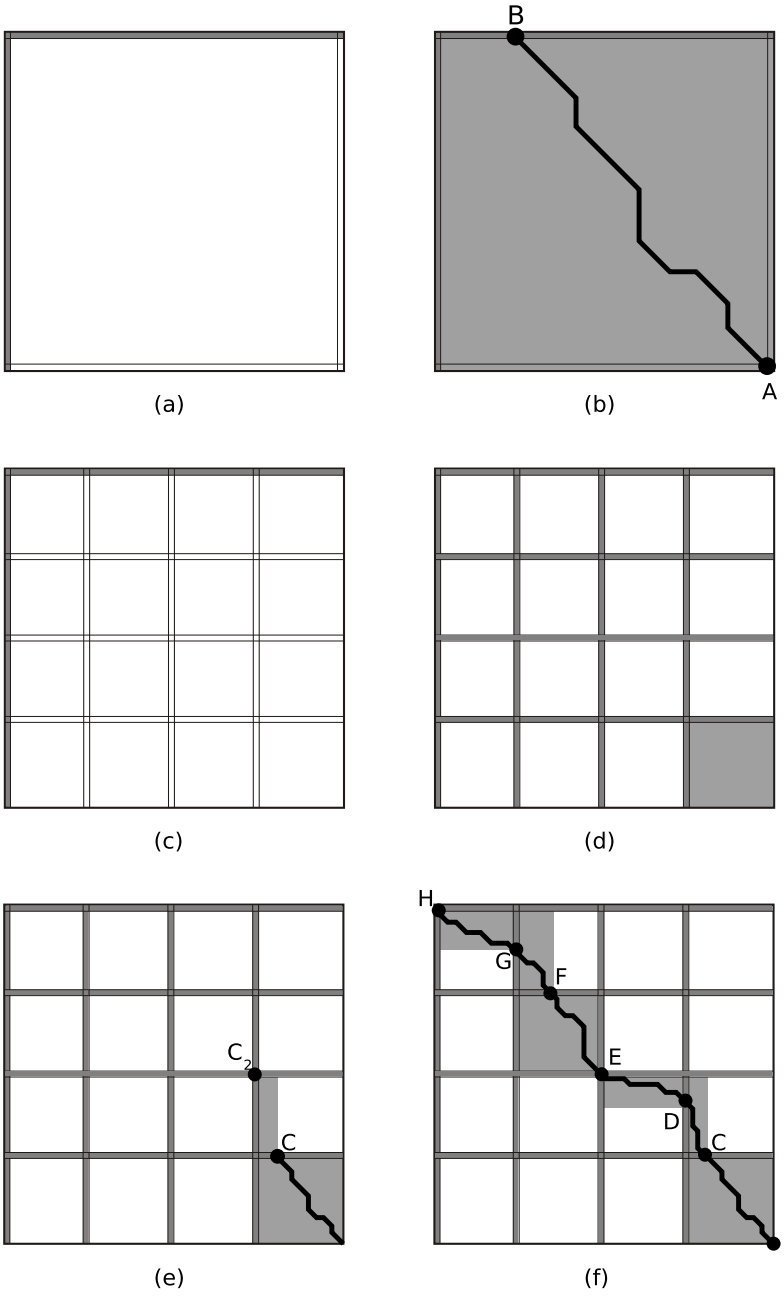


Figura 4.6: Fases do algoritmo FastLSA. [14]. Regiões em cinza indicam resultados calculados e guardados em memória. (a) Situação do *grid cache* no início do processamento. (b) *Base Case*: O alinhamento ótimo é encontrado preenchendo-se a matriz por completo. (c) *Grid cache* para $k = 4$. Alocado mas ainda não preenchido. (d) *Grid caches* preenchidos logo antes da recursão no último bloco processado. (e) Solução parcial encontrada. Próxima recursão logo após (d). (f) Alinhamento obtido por sucessivas recursões nos sub-problemas.

4.6f). O FastLSA opera, então, resolvendo uma sucessão de subproblemas retangulares, denominados *subproblemas FastLSA*. Se tomarmos $T(m, n, k)$ como sendo a quantidade de processamento realizado pelo algoritmo quando seqüências S e T são alinhadas utilizando um *grid cache* de k linhas e k colunas, temos que , no pior caso, $T(m, n, k) = \left(m \times n \times \frac{k+1}{k-1}\right)$[9]. Para $k = 5$, $T(m, n, 5) = 1,5 \times m \times n$, o que é inferior ao valor do algoritmo de Hirschberg, que vale sempre $2,0 \times m \times n$.

O FastLSA possui um desempenho superior ao Hirschberg devido a outro fator. Além da quantidade de processamento realizado, uma vez que a subdivisão em subproblemas que possam ser resolvidos utilizando-se uma memória de tamanho RM faz com que a localidade de *cache* seja maior.

O FastLSA foi paralelizado utilizando-se o processamento *wavefront*. Os processos de obtenção do *Base Case* e do *Fill Cache* são realizados dividindo-se estes em *tiles*, que são atribuídos ao processadores de acordo com a anti-diagonal. A implementação paralela foi realizada em um computador paralelo Origin 2400, com 32 processadores (CPUs 400 MHz R12000 MIPS), cada um com um *cache* de dados primário de 32KB e um *cache* secundário unificado de 8 MB. A implementação utilizou linguagem C, sobre o Irix 6.5 utilizando threads `sproc` em memória compartilhada em hardware.

As seqüências utilizadas nos testes com o algoritmo paralelo posuem aproximadamente 37 Kbp, 55 Kbp e 300 Kbp. Para 8 processadores, o algoritmo obtém um *speedup* bem próximo do linear. Para 16 e 32 processadores, o *speedup* obtido chega a ser pior que o obtido com 8 processadores.

4.6 GenomeDSM

Boukerche *et. al.*[7, 5, 35] avaliaram o problema de obter os alinhamentos locais de seqüências utilizando-se o paradigma de Memória Compartilhada Distribuída implementado por software pelo sistema denominado JIAJIA[25].

O algoritmo utilizado para obtenção dos alinhamentos é uma versão modificada do Smith-Waterman, utilizando heurísticas propostas por Martins[33]. Para cada valor de alinhamento da matriz, são armazenados também as coordenadas inicial e final do alinhamento, valores de máximo e de mínimo, número de *gaps*, contadores de *matches* e *mismatches* e um *flag* que indica se o alinhamento em questão é candidato a alinhamento ótimo.

Os valores de mínimo e de máximo são sempre atualizados com base no valor atual sendo calculado. A coordenada inicial é atualizada quando o *flag* de candidato possui valor 0 e o valor do alinhamento atinge um valor maior que um limiar estipulado pelo usuário, chamado valor de abertura do alinhamento. Neste caso, o valor da *flag* é também atualizado, passando a ter o valor 1, indicando que este alinhamento é candidato a um alinhamento ótimo. A coordenada final do alinhamento é obtida de forma semelhante. Quando a *flag* possui valor 1 e o valor do alinhamento cai abaixo do valor de máximo menos outro parâmetro estipulado pelo usuário, que é o valor de fechamento do alinhamento, a coordenada final do alinhamento é atualizada. Os valores deste alinhamento são copiados para a fila *alinhamentos* e a *flag* tem seu valor atualizado novamente para 0.

Os contadores de *matches*, *mismatches* e *gaps* são empregados quando o va-

lor sendo calculado foi obtido a partir de mais de uma entrada da matriz de similaridades, o que indica a presença de mais de um alinhamento ótimo. Para decidir entre os alinhamentos possíveis, é utilizada a expressão $(2 * matches + 2 * mismatches + gaps)$. Desta forma, os *gaps* são penalizados e os *matches* e *mismatches* são priorizados. A entrada da matriz que possuir o maior valor será a escolhida para fazer parte do alinhamento sendo calculado.

Se, ainda assim, os valores forem os mesmos, o critério de desempate é escolher o valor da esquerda, de cima e da diagonal, nesta ordem. Esta ordem tem por finalidade tentar aumentar a quantidade de *gaps* consecutivos no alinhamento. Ao final da execução do algoritmo, a fila de alinhamentos é ordenada pelo tamanho das subseqüências alinhadas e as entradas repetidas são removidas.

Os alinhamentos locais são obtidos acessando-se a fila de alinhamentos e então os valores de iniciais e finais dos alinhamentos indicam as subseqüências sobre as quais será executado um alinhamento global, produzindo-se assim os alinhamentos propriamente ditos.

Outra característica da heurística utilizada é utilizar espaço $O(n)$, pois são calculados os valores da matriz de similaridades utilizando sempre duas linhas de comprimento n. O tempo de execução do algoritmo continua $O(n^2)$.

A implementação paralela deste algoritmo foi chamada GenomeDSM. O GenomeDSM opera em duas fases, produzindo na primeira fase as posições iniciais e finais dos alinhamentos, e na segunda fase são efetivamente produzidos os alinhamentos. Utilizou-se o sistema de Memória Compartilhada Distribuída por software chamado JIAJIA[25]. Uma das características fundamentais deste sistema é utilizar o modelo de Consistência de Escopo[28]. O objetivo da Consistência do Escopo é aumentar o desempenho dos sistemas DSM que o empregam, limitando a comunicação e sincronização a escopos de consistência delimitados por operações de *lock* definidas pelo programador.

Também para este algoritmo, foi utilizado um método *wavefront*. A Figura 4.7 mostra como é distribuído o trabalho entre os processadores. A matriz é dividada igualmente em colunas entre os processadores, sendo que a comunicação entre os mesmos é feita através de colunas especiais que se encontram em memória compartilhada. Cada uma destas colunas especiais é compartilhada por dois processadores, onde um processador só grava dados na coluna e o outros só recupera dados desta mesma coluna.

A sincronização é feita através de *locks* e de variáveis de condição. Assim, enquanto o processador P_0 processa a primeira linha da sua sub-matriz, o processador, P_1 encontra-se esperando em uma variável de condição. Ao processar uma linha, P_0 grava os dados na coluna especial correspondente a P_1 e sinaliza através de uma variável de condição que este pode continuar a processar a linha.

Na primeira fase, à medida que os alinhamentos locais vão sendo encontrados pelos processadores, estes vão sendo gravados na fila *alinhamentos*. Na implementação paralela, a fila *alinhamentos* é acessada pelos processadores seguindo um mapeamento distribuído, ou seja, cada processador i armazena dados nas posições $i, i + p, i + 2p, \ldots$. Desta forma, não são necessárias operações de sincronização entre os processadores para que estes acessem a fila *alinhamentos*.

O GenomeDSM foi implementado em um *cluster* dedicado de 8 Pentium III 350 MHz com 160 MB de memória RAM, conectados por um *switch* Ethernet

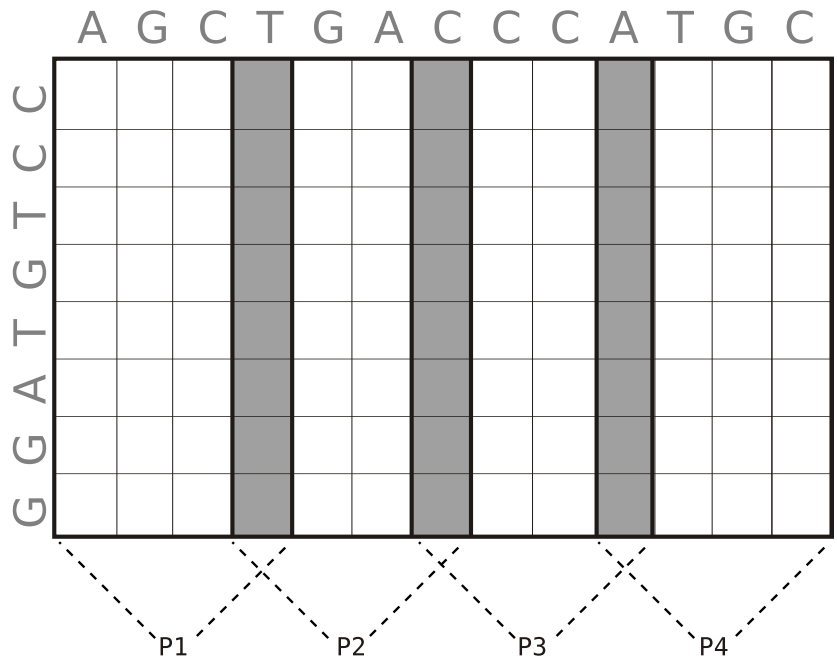


Figura 4.7: Distribuição do trabalho no GenomeDSM.[7]. As regiões em cinza indicam as áreas armazenadas em memória compartilhada.

100 Mbps. O sistema DSM utilizado foi o JIAJIA, rodando sobre Debian Linux 2.1. Para testar o desempenho, foram utilizadas seqüências de aproximadamente 15 Kbp, 50 Kbp, 80 Kbp, 150kbp e 400 Kbp. Os speedups calculados levam em conta o tempo total de execução, incluindo os tempos de inicialização e coleta de resultados. O melhor resultado obtido foi o de 4.58 para o alinhamento de seqüências de 400 Kbp em 8 processadores.

A segunda fase do algoritmo paralelo lê a fila *alinhamentos* utilizando também mapeamento distribuído, diminuindo assim a sincronização necessária nesta fase. Nesta fase, os *speedus* são melhores que os obtidos na primeira fase, sendo obtido um *speedup* de 7.57 para efetuar aproximadamente 1000 alinhamentos locais entre subseqüências utilizando 8 processadores.

Além dos testes de desempenho, foram comparados os resultados produzidos pelo GenomeDSM com os resultados obtidos utilizando-se o BLAST, FASTA e PipMaker. Para o alinhamento de seqüências de aproximandamente 50 Kbp os resultados obtidos pelo GenomeDSM foram condizentes com os obtidos pelo BLAST e pelo PipMaker. O FASTA falhou em encontrar alguns dos alinhamento devido a uma limitação que este algoritmo possui quanto ao comprimento das seqüências que pode analisar.

4.7 Estratégias paralelas em CoW (*Cluster of Workstations*)

Boukerche *et. al.*[6] publicaram um estudo avaliando três estratégias propostas para rodar o algoritmo de Smith-Waterman em um *cluster* de estações de

trabalho. A primeira estratégia comentada é a publicada em [7], denominada GenomeDSM, citada na Seção 4.6. A segunda é uma otimização do GenomeDSM, denominada *heuristic-block*. A terceira estratégia é uma abordagem exata, ao contrário das duas primeiras que utilizam heurísticas. Todas as três estratégias foram implementadas utilizando o JIAJIA 2.1 como sistema de memória compartilhada distribuída por *software*.

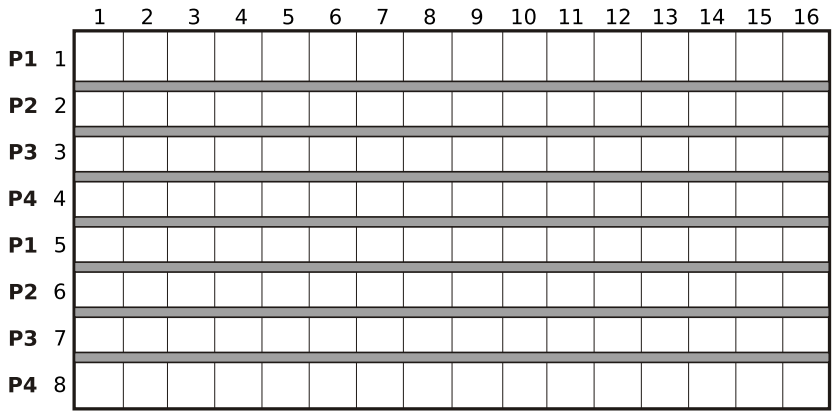


Figura 4.8: Distribuição de trabalho em blocos e bandas de passagem[6]. As regiões em cinza motram os trechos da matriz de similaridades que são comunicados entre os processadores.

A estratégia *heuristic-block* trabalha a granularidade do GenomeDSM. Neste último, a cada linha da matriz de similaridades, cada processador efetua uma operação de *lock* e *unlock* em um segmento de memória compartilhada. Assim, para otimizar a obtenção de resultados, a estratégia *heiristic-block* subdivide o trabalho em blocos, usando uma estratégia semelhente à descrita na Seção 4.4, o que é ilustrado também na Figura 4.8. A comunicação só é realizada, quando um bloco, cujo tamanho é definido pelo usuário, tem todos os seus valores processados. O tamanho dos blocos tem papel importante no desempenho do algoritmo.

Os resultados obtidos com esta estratégia são bem satisfatórios. Por exemplo, a obtenção de alinhamentos para seqüências de aproximadamente 50 Kbp que demora 1.107,02s na estratégia *heuristics* (GenomeDSM), demora 313,13s na estratégia *heuristic-block*.

Tal ganho se deve principalmente a dois fatores. Primeiramente, o uso de blocos no processamento da matriz favorece a localidade de *cache*, aumentando significativamente o desempenho do algoritmo, mesmo quando executado em apenas um processador. Em segundo lugar, a quantidade de operações de *lock* e *unlock* diminui consideravelmente, uma vez que estas operações só são realizadas uma vez a cada bloco.

A terceira estratégia proposta utiliza o algoritmo de Smith-Waterman sem heurísticas para pré-processar a matriz de similaridades. O pré-processamento calcula os valores da matriz de similaridade utilizando bandas como na Figura 4.8. A Figura 4.9 mostra em mais detalhes a estrutura de cada banda no caso da terceira estratégia. Entre cada faixa, existe uma *banda de passagem*, que fica situada em memória compartilhada e também é armazenada em disco para posterior utilização na obtenção dos alinhamentos. Assim como na segunda estratégia,

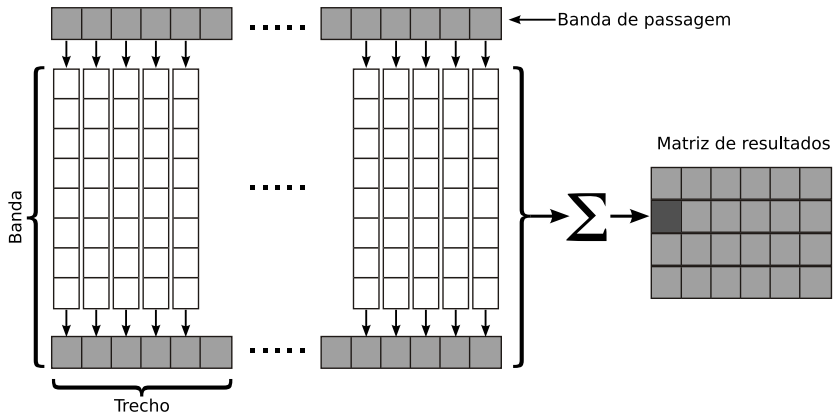


Figura 4.9: Bandas de passagem. A estrutura de dados utilizada na terceira estra

os dados são primeiramente processados em blocos para só então serem comunicados ao próximo processador. A largura de cada bloco define um *trecho* sobre a *banda de passagem*. A menor quantidade de comunicação efetuada corresponde à transmissão de um *trecho* entre processadores.

Além do armazenamento em disco das bandas de passagem, a estratégia utiliza uma *matriz de resultados* onde, para cada coluna ou conjunto de colunas de uma banda, são contados o número de elementos da matriz que possuem um valor maior que um limiar estipulado pelo usuário, sendo este número denominado *acertos*. Este contador serve para indicar em quais regiões da matriz de similaridades é mais provável encontrar os melhores alinhamentos.

A estratégia utiliza como parâmetros, dentre outros, a altura das bandas em linhas, o tamanho do bloco a ser processado, a função de crescimento do bloco e a quantidade de colunas utilizadas para se sumarizar os *acertos* na *matriz de resultados*. A função de crescimento, neste caso, indica se todos os blocos terão largura constante, ou se esta largura aumenta à medida que as banda vão sendo processadas.

Os testes realizados com a terceira estratégia resultaram em bons *speedups*, onde o melhor caso foi de 6,8 para 8 processadores para uma seqüência de 80 Kbp. A *matriz de resultados* mostrou que é capaz de indicar regiões da matriz de similaridades onde podem ser encontrados os melhores alinhamentos.

Todos os testes realizados com as estratégias citadas foram feitos sobre a mesma plataforma descrita na Seção 4.6. Além disso, todas as estratégias têm foco na obtenção de alinhamentos locais, possuindo complexidade de espaço $O(n)$.

Além das estratégias citadas, o artigo faz uma observação teórica a respeito de como pode ser diminuído o custo de espaço para $O(min[n, m] + n'^2)$ durante a obtenção de alinhamentos locais. Neste caso, n'^2 é o comprimento do maior alinhamento encontrado para as seqüências S e T. Tal idéia está fundamentada no fato de que, dadas as seqüências s e t, se há um alinhamento de valor s terminando em (i, j), então, para as seqüências invertidas s^{inv} e t^{inv}, existe um alinhamento de valor s, iniciando-se em $(n - i + 1, m - j + 1)$. Baseando-se nesta observação, o algoritmo descrito na Figura 4.10 é proposto:

Algoritmo de espaço reduzido
Entrada:Seqüências S e T

Executar o argoritmo SW sobre s e t usando espaço linear (Utilizar apenas um array $O(n)$ para tanto);

Para cada alinhamento desejado de valor k encerrando em i, j:

Executar o algoritmo em programação dinâmica sobre as seqüências $s[1 \ldots i]^{inv}$ e $t[1 \ldots j]^{inv}$ até encontrar um alinhamento de valor k;

Reconstruir o alinhamento dos reversos sobre as seqüências iniciais.

Figura 4.10: Algoritmo para redução da complexidade de espaço.

4.8 Quadro comparativo

A Tabela 4.2 mostra uma comparação feita entre as soluções descritas neste Capítulo. São citadas informações acerca dos tipos de alinhamentos realizados, quantidade de alinhamentos produzidos, funções de *gap* utilizadas, complexidade dos algoritmos e a plataforma onde foram implementadas as soluções.

Alinhamentos	Prefixo Paralelo	Modelo EARTH	PSW-DC	Block-cyclic Wavefront	FastLSA	GenomeDSM	Estratégias em CoW
Global	•	•			•		
Local	•	•	•	•		•	•
Quantidade.	N/C	Vários	1	N/C	1	Vários	Vários
Função de *gap*							
Constante	•	•	•	•	•	•	•
Linear	•				•		
Complexidade							
Tempo	$O(\frac{mn}{p})$	N/C	$O(\frac{mn}{p})$	$O(\frac{mn}{p})$	$O(\frac{mn}{p})$	$O(\frac{mn}{p})$	$O(\frac{mn}{p})$
Espaço	$O(m+\frac{n}{p})$	N/C	$O(\frac{mn}{p})$	$O(m+n)$	$O(m+n)$	$O(m+n)$	$O(m+n)$
Implementação							
Processadores	N/C	128	16	16	32	8	8
Processador	N/C	Pentium Pro 200MHz	Dawning 2000-I	Pentium IV Xeon 2.6GHz	SGI Origin 2400	Pentium III 350MHz	Pentium III 350MHz
Memória	N/C	128MB	N/C	512MB	N/C	160MB	160MB
Rede	N/C	100Mbps	N/C	1Gbps Myrinet	N/C	100Mbps	100Mbps
Linguagem	N/C	Threaded-C	C	C++	C	C	C
Comunicação	N/C	EARTH	MPI	MPI	SM	DSM	DSM

N/A = Não se aplica
N/C = Não comentado na referência
MPI = Message Passing Interface
SM = Memória Compartilhada
DSM = Memória Compartilhada Distribuída

Tabela 4.2: Quadro comparativo das soluções paralelas.

Capítulo 5

Projeto da estratégia *z*Align

A estratégia a ser apresentada foi desenvolvida com o intuito de realizar o alinhamento local entre duas seqüências biológicas longas, usando um algoritmo exato e função linear de *gaps*(Seção 2.5) para a obtenção dos alinhamentos ótimos.

Serão descritas adiante as técnicas e adaptações que foram feitas em algoritmos já descritos nos Capítulos 2 e 4 a fim de se produzir alinhamentos ótimos utilizando-se comparações exatas em tempo hábil e em espaço restrito de memória.

5.1 Estratégia proposta

A estratégia proposta, referida doravante como *z*Align, trabalha em quatro fases. Na primeira fase, é feita uma distribuição de dados para todos os processadores. Na segunda fase, ocorre o processamento da matriz de similaridades para o inverso das seqüências de entrada S e T, sendo produzidas informações sobre os alinhamentos ótimos locais encontrados em cada processador. Na terceira fase, estes dados são agregados e são descartados os alinhamentos que não possuem *score* máximo. Na quarta fase, a partir das informações encontradas na segunda fase, são produzidos os alinhamentos ótimos entre S e T.

5.1.1 1ª Fase: Distribuição

A primeira fase da estratégia é responsável por distribuir entre os processadores os dados que estes irão utilizar como entrada.

Os dados distribuídos são as seqüências de DNA S e T a serem comparadas, os valores que serão atribuídos à ocorrência de *matches*, *mismatches*, aberturas de *gap* e extensão de *gaps* e as informações de sub-divisão da matriz de similaridades. Sendo assim, ao final desta fase, estes dados encontram-se copiados na memória de cada processador que irá participar do alinhamento.

5.1.2 2ª Fase: Programação dinâmica

5.1.2.1 Divisão dos dados e processamento

Na fase de programação dinâmica, é utilizado um algoritmo de Gotoh[22] modificado para a produção de alinhamentos usando uma função linear de *gaps*. Além disso, são calculados valores de divergência para os alinhamentos, sendo estes usados posteriormente ao se produzir os alinhamentos propriamente ditos. A obtenção destes valores de divergência e o uso dos mesmos são descritos mais adiante na Seção 5.1.2.2.

Esta fase envolve o cálculo da matriz de similaridades $S_{i,j}$, das matrizes de gaps $P_{i,j}$ e $Q_{i,j}$ e das matrizes de divergência $Div_{i,j}^{sup}$ e $Div_{i,j}^{inf}$. Todas estas matrizes são calculadas utilizando-se as seqüências inversas S^{inv} e T^{inv}. A justificativa para o uso das seqüências inversas será dada na Seção 5.1.4.

A fim de simplificar a descrição dos passos do **z**Align, será mostrada apenas a divisão da matriz $S_{i,j}$ entre os processadores. A divisão das matrizes $P_{i,j}$, $Q_{i,j}$, $Div_{i,j}^{sup}$ e $Div_{i,j}^{inf}$ ocorre de forma análoga.

A matriz $S_{i,j}$ é dividida verticalmente em sub-matrizes de mesmo tamanho, ficando cada processador responsável por uma destas sub-matrizes. Tomando-se m e n como o comprimento das seqüências comparadas e p o número de processadores sendo utilizados, cada processador fica então responsável por calcular os valores de uma sub-matriz de tamanho $\frac{m}{p} \times n$.

Cada sub-matriz é então novamente dividida verticalmente e horizontalmente, formando uma matriz de blocos. Assim, definindo h como o número de colunas e v como o número de linhas em que as sub-matrizes foram divididas, cada bloco possui tamanho $\frac{m}{p \times h} \times \frac{n}{v}$. A Figura 5.1 ilustra a divisão dos dados.

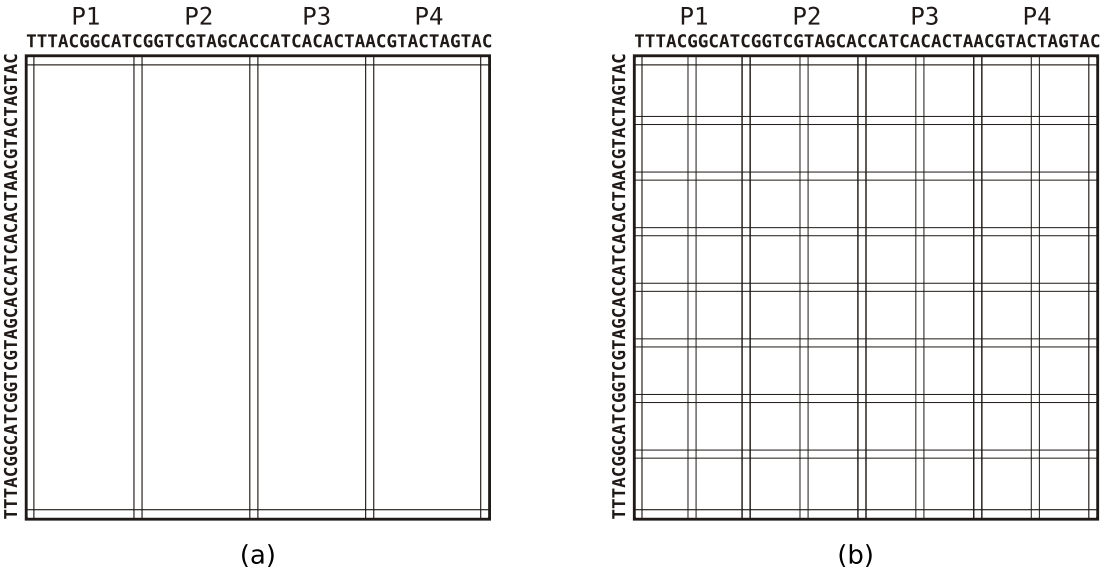


Figura 5.1: Divisão da matriz de similaridades entre os processadores. (a) Distribuição de $S_{i,j}$ entre 4 processadores. (b) Divisão em blocos tomando-se $h = 2$ e $v = 8$.

Os blocos estão dispostos de forma que cada dois blocos adjacentes compartilham entre si uma mesma linha ou coluna de $S_{i,j}$. Estas linhas e colunas

compartilhadas são denominadas linhas (ou colunas) de passagem. Cada bloco possui então duas linhas e colunas de passagem, sendo que a primeira linha e primeira coluna servem como entrada de dados e a última linha e última coluna servem como saída de dados. A Figura 5.2 ilustra estas definições.

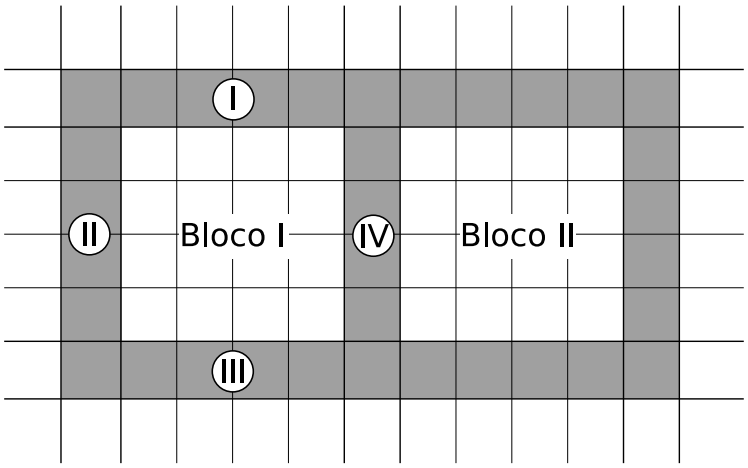


Figura 5.2: Linhas e colunas de passagem. A Linha I e a Coluna II contêm os dados necessários para processar o Bloco I. Tendo sido o Bloco I completamente processado, a Linha III e a Coluna IV são utilizados como entrada para outros blocos. A Coluna IV é ao mesmo tempo coluna de saída do Bloco I e coluna de entrada do Bloco II.

Para tornar viável a comparação de seqüências longas, os valores de $S_{i,j}$ não são guardados inteiramente na memória. Os blocos são processados da esquerda para a direita, sendo que uma linha inteira de blocos deve ser processada para que se inicie o processamento da linha de blocos seguinte. Em memória são armazenadas apenas as colunas e linhas de passagem necessárias para se processar a linha de blocos atual. Quando uma linha inteira de blocos é processada, sua linha de saída é utilizada como linha de entrada da próxima linha de blocos.

O processamento de cada bloco é também feito utilizando-se um espaço linear de memória. A técnica utilizada é a mesma descrita na Seção 2.6.1, onde são utilizadas apenas duas linhas para produzir os valores de $S_{i,j}$. A partir da linha e da coluna de entrada, os valores de similaridade vão sendo obtidos, linha por linha ,e vão sendo descartados à medida que não são mais necessários para se produzir diretamente outros valores de similaridade.

O objetivo da estratégia *z*Align é obter os alinhamentos que possuam o melhor *score*. Assim, uma lista é guardada com todas as informações pertinentes aos melhores alinhamentos locais encontrados em cada processador. Todos os alinhamentos presentes nesta lista possuem o mesmo *score*. Toda vez que um alinhamento encontrado possui um *score* maior que o dos alinhamentos na lista, esta é esvaziada e o novo alinhamento passa a ser o único alinhamento listado. Se um alinhamento encontrado possui o mesmo *score* que o dos alinhamentos da

lista, este é adicionado à lista. Alinhamentos encontrados cujo *score* seja inferior ao *score* de entrada da lista são ignorados.

Para cada alinhamento na lista, são armazenados a posição (i, j) de término do alinhamento, o *score* obtido e os valores de divergência superior e inferior encontrados. A partir destas informações, é possível reconstruir os alinhamentos encontrados. Esta reconstrução é descrita em detalhes na quarta fase.

Devido à dependência de dados imposta pelo uso do algoritmo de Gotoh (Seção 2.5.1), um bloco só pode ser processado se a sua linha e coluna de entrada já tiverem sido preenchidas com os valores corretos. Como cada célula da matriz de similaridades depende diretamente das células acima e à esquerda, esta mesma dependência acaba por se repetir nos blocos. Assim, o processamento dos blocos irá obedecer a um padrão em onda (*wavefront*). Vale ressaltar que a primeira linha e coluna da matriz de similaridades já possuem os seus valores definidos pelo algoritmo de Gotoh durante a sua inicialização, satisfazendo a dependência da primeira linha de blocos.

A Figura 5.3 mostra como o processamento em onda ocorre nos blocos. Logo no início, apenas P1 pode processar a primeira linha de blocos. P2, P3, e P4 estão aguardando o preenchimento dos valores das colunas de entrada de seus primeiros blocos, o que é ilustrado na Figura 5.3a. Na Figura 5.3b, P1 acaba de terminar de processar a primeira linha de blocos. P1 envia a P2 os valores contidos na coluna de passagem de saída do último bloco processado.

A Figura 5.3c mostra P1 processando a segunda linha de blocos e P2 continuando o processamento da primeira linha de blocos . O processamento continua até que P2 tenha terminado a sua primeira linha de blocos e P1 tenha terminado a sua segunda linha de blocos. O processamento continua desta forma e na Figura 5.3d, vê-se claramente a frente de onda de processamento.

Os valores de h e v possuem um impacto considerável no desempenho do ***z***Align. O valor de v define o número de vezes que os processadores irão se comunicar, bem como a quantidade de dados envolvidos em cada comunicação. Como a comunicação só ocorre quando um processador completa o processamento de uma linha de blocos, blocos muito "altos" podem fazer com que, no início e no final do processamento, muito tempo seja consumido sem que todos os processadores estejam ocupados.

Já blocos com poucas linhas podem aumentar muito o custo de comunicação, pois o custo de envio de cada mensagem depende não somente do tamanho da mesma, mas também de um custo associado ao próprio envio da mensagem pela rede de intercomunicação. Além disso, uma vez que as comunicações nesta fase são realizadas de forma síncrona, deve-se evitar realizar um grande número de comunicações, devido também ao próprio custo da sincronização.

O valor de h define a "largura" dos blocos. O processamento dos blocos utiliza apenas duas linhas para obter os valores de similaridade contidos no bloco, sendo que estas linhas têm o mesmo tamanho da largura do bloco. Uma boa escolha para h permite manter as duas linhas inteiras dentro das linhas de *cache* do processador, o que tem um impacto muito positivo na eficiência do processamento.

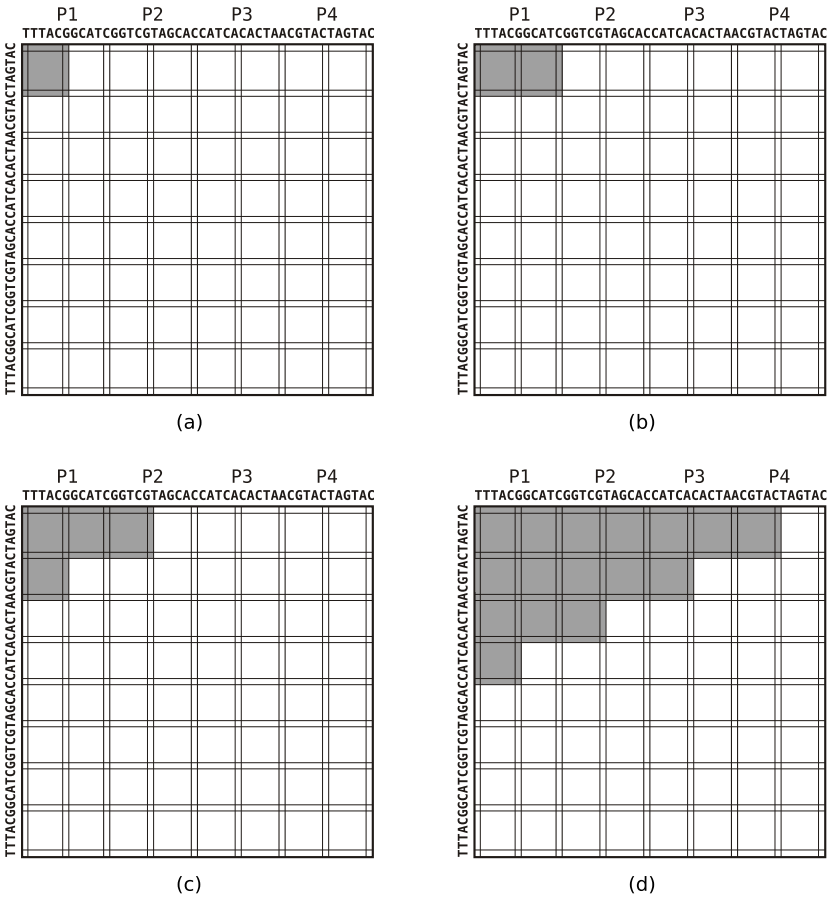


Figura 5.3: Ordem de processamento dos blocos para $p = 4$, $h = 2$ e $v = 8$. As áreas escuras representam os blocos já processados.

5.1.2.2 Cálculo da divergência

O algoritmo original de Gotoh[22] utiliza-se de várias matrizes para produzir os alinhamentos locais ótimos entre duas seqüências. Estas matrizes armazenam os valores de similaridade $S_{i,j}$, as "direções" $e_{i,j}$ encontradas, valores $P_{i,j}$ de *gaps* em S e valores $Q_{i,j}$ de *gaps* em T.

Durante a segunda fase do ***z***Align, é utilizada uma versão modificada do algoritmo de Gotoh onde a matriz $e_{i,j}$ não é calculada, pois a obtenção dos alinhamentos é realizada somente na quarta fase.

As modificações efetuados no algoritmo de Gotoh residem no cálculo de mais duas matrizes $Div_{i,j}^{sup}$ e $Div_{i,j}^{inf}$, contendo as divergências máximas superior e inferior. Os valores de divergência são definidos a seguir:

Seja $\mathcal{A}$ um alinhamento local entre duas seqüências S e T. Este pode ser definido como uma lista de pares ordenados $\mathcal{A} = ((i_1, j_1), (i_2, j_2), \ldots, (i_{|\mathcal{A}|}, j_{|\mathcal{A}|}))$, onde $1 \leq i_k \leq |S|$ e $1 \leq j_k \leq |T|$ para $1 \leq k \leq |\mathcal{A}|$. Além disso, $0 \leq i_k - i_{k-1} \leq 1$ e $0 \leq j_k - j_{k-1} \leq 1$, para $2 \leq k \leq |\mathcal{A}|$. Para um alinhamento local $\mathcal{A}$, também vale que $(i_k, j_k) \neq (i_{k-1}, j_{k-1})$, para $2 \leq k \leq |\mathcal{A}|$.

A lista $\mathcal{A}$ representa então as posições da matriz de similaridades que fazem parte do alinhamento. A partir de $\mathcal{A}$ podemos definir uma lista $\mathcal{A}' = (a'_1, a'_2, \ldots, a'_{|\mathcal{A}|})$ onde $a'_k = i_k - j_k$, para $1 \leq k \leq |\mathcal{A}|$. Esta lista $\mathcal{A}'$ representa as diagonais sobre as quais o alinhamento $\mathcal{A}$ é definido.

Definindo-se $a'_{sup} = min[a'_1, a'_2, \ldots, a'_{|\mathcal{A}|}]$ e, analogamente, definindo-se $a'_{inf} = max[a'_1, a'_2, \ldots, a'_{|\mathcal{A}|}]$, então podemos obter $Div^{sup}(\mathcal{A}) = a'_{sup} - a'_{|A'|}$, bem como $Div^{inf}(\mathcal{A}) = a'_{inf} - a'_{|A'|}$.

A Figura 5.4 ilustra as definições anteriores. Na Figura 5.4a é mostrada uma representação de um alinhamento local entre duas seqüências. Neste caso, $\mathcal{A} = ((1,2),(2,3),(2,4),(3,5),(4,5),(5,5),(6,6),(6,7),(7,8))$. Na Figura 5.4b, são mostrados os valores associados às diagonais, sendo neste caso $\mathcal{A}'$ =(-1, -1, -2, -2, -1, 0, 0, -1, -1). Logo $a'_{sup} = -2$ e $a'_{inf} = 0$. Como $a'_{|\mathcal{A}'|} = -1$, então $Div^{sup}(\mathcal{A}) = (-2) - (-1) = -1$ e $Div^{inf}(\mathcal{A}) = (0) - (-1) = 1$

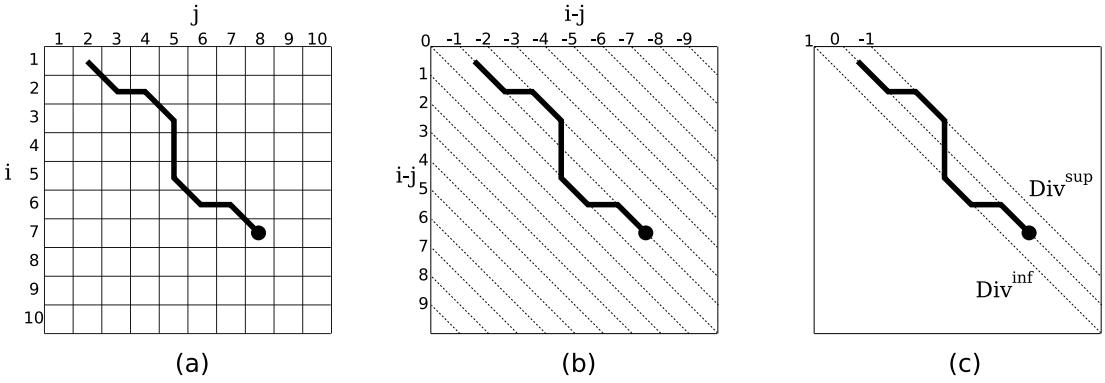


Figura 5.4: Alinhamento, diagonais e divergências. (a) Representação de um alinhamento local. (b) Diagonais da matriz $S_{i,j}$ e seus valores. (c) Divergências superior e inferior.

Uma vez definido o conceito de divergência superior e inferior, resta mostrar como o ***z***Align obtém estes valores a partir da matriz de similaridades. Seja $S_{i,j}$

a matriz de similaridades, $\mathrm{Div}_{i,j}^{sup}$ a matriz de divergências superiores e $\mathrm{Div}_{i,j}^{inf}$ a matriz de divergências inferiores.

Para cada elemento de $\mathrm{Div}_{i,j}^{sup}$ e de $\mathrm{Div}_{i,j}^{inf}$, é aplicado o algoritmo descrito na Figura 5.5 que procede da seguinte forma. Se $\mathrm{S}_{i,j} = 0$, então os valores de divergência são o próprio valor da diagonal no ponto (i, j), onde está se iniciando um novo alinhamento local. Caso $\mathrm{S}_{i,j} > 0$, então o que há na verdade é a continuação de um alinhamento local. O algoritmo então verifica a a partir de que valores $\mathrm{S}_{i,j}$ se originou, podendo ser de $\mathrm{S}_{i-1,j-1}$, de $\mathrm{S}_{i-1,j}$ e de $\mathrm{S}_{i,j-1}$. Para os valores que deram origem a $\mathrm{S}_{i,j}$, obtém-se os respectivos máximos e mínimos de divergência. Assim, para cada alinhamento obtido, sabe-se as diagonais máxima e mínima que definem a menor faixa diagonal da matriz de similaridades na qual o alinhamento se restringe.

Algoritmo Cálculo de divergência
$\mathrm{Div}_{i,j}^{sup} \leftarrow i - j$
$\mathrm{Div}_{i,j}^{inf} \leftarrow i - j$
if $\mathrm{S}_{i,j} > 0$ **then**
  if $\mathrm{S}_{i,j} = \mathrm{S}_{i-1,j-1}$ **then**
    $\mathrm{Div}_{i,j}^{sup} \leftarrow min[\mathrm{Div}_{i,j}^{sup}, \mathrm{Div}_{i-1,j-1}^{sup}]$
    $\mathrm{Div}_{i,j}^{inf} \leftarrow max[\mathrm{Div}_{i,j}^{inf}, \mathrm{Div}_{i-1,j-1}^{inf}]$
  if $\mathrm{S}_{i,j} = \mathrm{S}_{i-1,j}$ **then**
    $\mathrm{Div}_{i,j}^{sup} \leftarrow min[\mathrm{Div}_{i,j}^{sup}, \mathrm{Div}_{i-1,j}^{sup}]$
    $\mathrm{Div}_{i,j}^{inf} \leftarrow max[\mathrm{Div}_{i,j}^{inf}, \mathrm{Div}_{i-1,j}^{inf}]$
  if $\mathrm{S}_{i,j} = \mathrm{S}_{i,j-1}$ **then**
    $\mathrm{Div}_{i,j}^{sup} \leftarrow min[\mathrm{Div}_{i,j}^{sup}, \mathrm{Div}_{i,j-1}^{sup}]$
    $\mathrm{Div}_{i,j}^{inf} \leftarrow max[\mathrm{Div}_{i,j}^{inf}, \mathrm{Div}_{i,j-1}^{inf}]$

Figura 5.5: Cálculo da divergência

5.1.3 3ª Fase: Agregação

Ao final da segunda fase, são conhecidos as divergências, o *score* e a posição final em S^{inv} e T^{inv} de cada alinhamento local ótimo para cada processador. O trabalho da terceira fase é criar uma lista unificada de todos os valores encontrados.

Cada processador envia a sua lista de alinhamentos para o processador P_1. Este, por sua vez, verifica quais das listas possuem o *score* máximo. As listas que não possuírem *score* máximo são descartadas, restando apenas as listas com valores referentes aos alinhamentos locais ótimos.

Ao final da terceira fase, as listas que sobraram são concatenadas para formar uma lista com todos os valores dos alinhamentos locais ótimos.

5.1.4 4ª Fase: Alinhamento

É na quarta fase do ***z***Align que os os alinhamentos são realmente produzidos a partir da lista unificada obtida na terceira fase.

A partir deste momento, o ***z***Align passa a ter um comportamento cliente-servidor. P_1 passa a receber requisições dos outros processadores, respondendo a cada requisição com os dados de um alinhamento da lista. O processador que fez a requisição recebe estes dados e produz o respectivo alinhamento. O alinhamento produzido é então comunicado a P_1, que exibe na saída padrão os seus dados.

É fato que um alinhamento local $\mathcal{A} = ((i_1, j_1), (i_2, j_2), \ldots, (i_{|\mathcal{A}|}, j_{|\mathcal{A}|}))$ entre S e T corresponde a um alinhamento global entre $S' = S[i_1 \ldots i_{|\mathcal{A}|}]$ e $T' = T[j_1 \ldots j_{|\mathcal{A}|}]$. O que é feito na quarta fase é um alinhamento global para as subseqüências S' e T' pela execução de um algoritmo modificado de Fickett[15] para cada alinhamento local ótimo encontrado.

A primeira adaptação ao algoritmo de Fickett é a utilização de funções lineares de *gap*, para que o alinhamento global das subseqüências produza o mesmo resultado obtido com o alinhamento local das seqüências completas. Isto envolve utilizar, além da matriz $\mathrm{S}_{i,j}$, as matrizes $\mathrm{P}_{i,j}$, $\mathrm{Q}_{i,j}$ e $e_{i,j}$ do algoritmo de Gotoh[22].

A segunda modificação se deve ao fato de a segunda fase fornecer apenas a posição final (i^{inv}, j^{inv}) em S^{inv} e T^{inv} para um alinhamento $\mathcal{A}^{inv}$ encontrado. Primeiro, os valores de i^{inv} e j^{inv} são transformados nos valores i e j de início referentes ao mesmo alinhamento, só que sobre S e T. A Observação 5.1.1[6] justifica isso.

Observação 5.1.1 ***(Alinhamento sobre os inversos)*** *Suponha um alinhamento local de score k encerrando-se nas posições i e j entre as seqüências S e T. Então, existe um alinhamento de mesmo score iniciando-se nas posições $|S| - 1 + i$ e $|T| - j + 1$ das seqüências invertidas S^{inv} e T^{inv}.*

O que ocorre na verdade, é que a segunda fase realiza o alinhamento sobre os inversos S^{inv} e T^{inv}. Os pontos finais dos alinhamentos encontrados são exatamente os pontos de início destes mesmos alinhamentos em S e T.

Tendo os pontos iniciais (i, j) dos alinhamentos, estariam faltando ainda os pontos finais dos mesmos. No entanto, como a segunda fase também já calculou os *scores* dos alinhamentos, basta executar o algoritmo de Fickett sobre as seqüências $S[i \ldots |s|]$ e $T[j \ldots |t|]$ até que seja encontrada um valor de $\mathrm{S}_{i,j}$ igual ao *score* procurado. A partir deste ponto é feito o *traceback*, seguindo a matriz $e_{i,j}$ de direções para produzir o alinhamento. A Figura 5.6 ilustra este passo.

Realizar a segunda fase sobre S^{inv} e T^{inv} e a quarta fase sobre S e T se deve à forma como o ***z***Align foi testado para saber se este produzia corretamente os alinhamentos ótimos. Antes de se implementar o ***z***Align, o algoritmo de Gotoh[22] foi implementado como referência.

Uma implementação seqüencial foi feita para o ***z***Align, sendo que esta realizava a segunda fase sobre S e T e a quarta fase sobre S^{inv} e T^{inv}. Os alinhamentos obtidos para esta implementação eram comparados com os produzidos pela implementação de Gotoh. Todos os alinhamentos ótimos produzidos possuíam o mesmo *score* e posições de início e fim em S e T, mas os alinhamentos reportados eram diferentes. A diferença entre os alinhamentos era a disposição dos *matches*, *mismatches* e *gaps* ao longo dos mesmos.

Acontece que, para um *score* ótimo, entre seus pontos de início e de fim correspondentes, podem existir vários alinhamentos ótimos, dependendo de como o

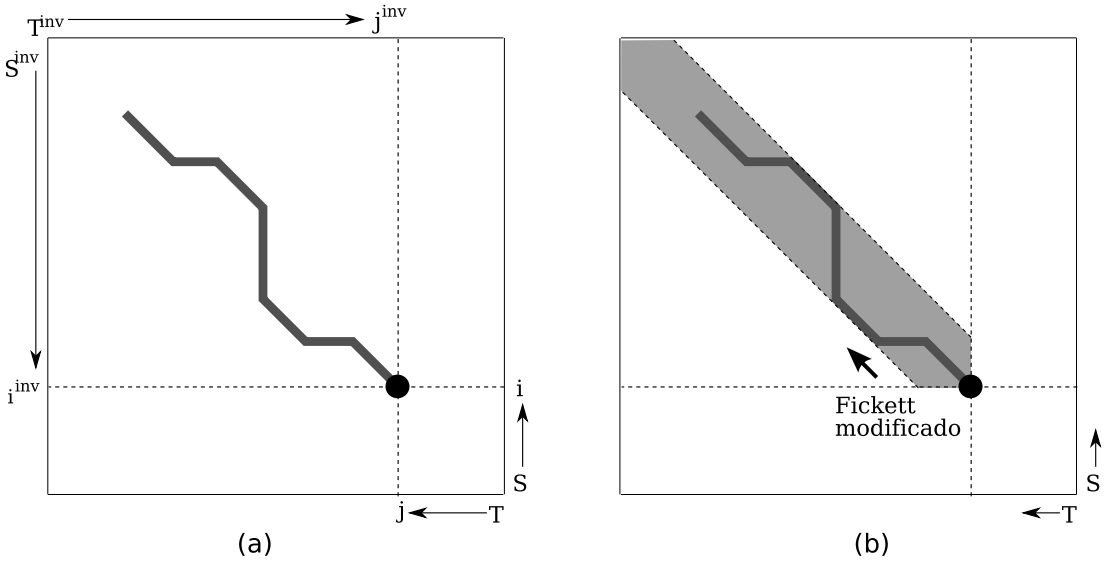


Figura 5.6: Alinhamento entre S e T e suas inversas. (a) Posição final de um alinhamento local entre S^{inv} e T^{inv} é igual ao ponto de início deste mesmo alinhamento em S e T. (b) A área cinza representa a região que poderá ser processada pelo algoritmo modificado de Fickett sobre S e T no sentido indicado. O alinhamento representado ainda não foi obtido.

traceback é realizado. Todo valor $S_{i,j}$ que tenha sido obtido a partir de mais de um dentre os valores $S_{i-1,j-1}$, $S_{i-1,j}$ e $S_{i,j-1}$ multiplica a quantidade de alinhamentos ótimos possíveis. O sentido de execução do *traceback* é um dos fatores que determinam qual dos alinhamentos ótimos será retornado, principalmente quando é retornado apenas o primeiro alinhamento ótimo encontrado em um *traceback*. A Figura 5.7 mostra um exemplo de como o sentido do *traceback* influi na obtenção dos alinhamentos ótimos.

Assim, para obter exatamente os alinhamentos retornados pela implementação de referência do algoritmo de Gotoh, optou-se por realizar a segunda fase sobre S^{inv} e T^{inv} para que o *traceback* realizado na quarta fase fosse feito no mesmo sentido do *traceback* feito pela implementação de referência de Gotoh. Os resultados produzidos para as duas implementações passaram a ser idênticos.

O algoritmo original de Fickett, para produzir alinhamentos, recebe como entrada as seqüências S e T, e um valor estimativa d para o score ótimo do alinhamento entre S e T. A partir destes dados, são estimadas as diagonais k e l que limitarão o processamento da matriz de similaridades. No caso do ***z***Align, já se sabe exatamente o valor do *score* do alinhamento, bem como os valores de k e l, que neste caso são Div^{inf} e Div^{sup} respectivamente.

A terceira modificação envolve armazenar em memória apenas a faixa da matriz de similaridades restringida por Div^{inf} e Div^{sup}. Além disso, a memória vai sendo alocada à medida que o processamento avança pela matriz de similaridades. Desta forma, tanto o processamento quanto o uso de memória é restringido aos valores estritamente necessários para se obter um alinhamento. A Figura 5.8 ilustra este conceito.

A quarta modificação parte do princípio que, mesmo de posse das divergências

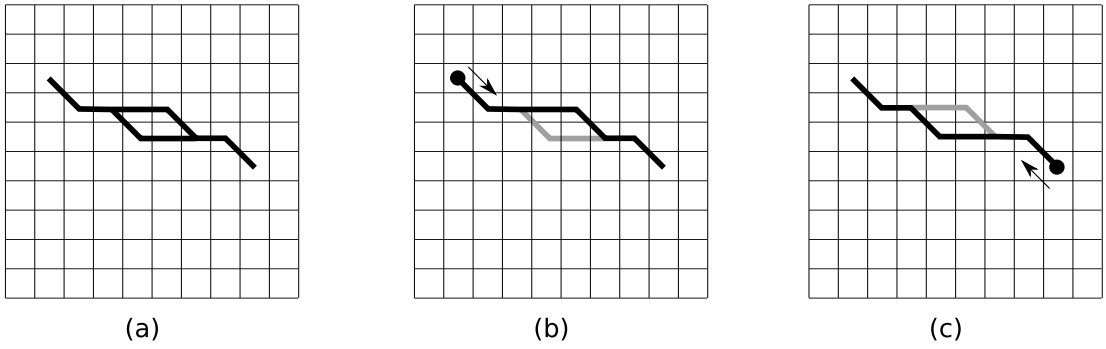


Figura 5.7: Múltiplos alinhamentos ótimos. (a) Exemplo de dois alinhamentos ótimos com mesmo ponto de início e de fim. (b) O alinhamento para S^{inv} e T^{inv} produz um *traceback* no sentido indicado. Alinhamentos com funções lineares de *gap* tendem a produzir a maior quantidade de *gaps* consecutivos, assim neste caso o *traceback* não escolhe o caminho inferior na primeira bifurcação. (c) O alinhamento para S e T produz um *traceback* no sentido inverso, fazendo com que o alinhamento obtido neste caso seja diferente, mesmo sendo este um alinhamento ótimo também.

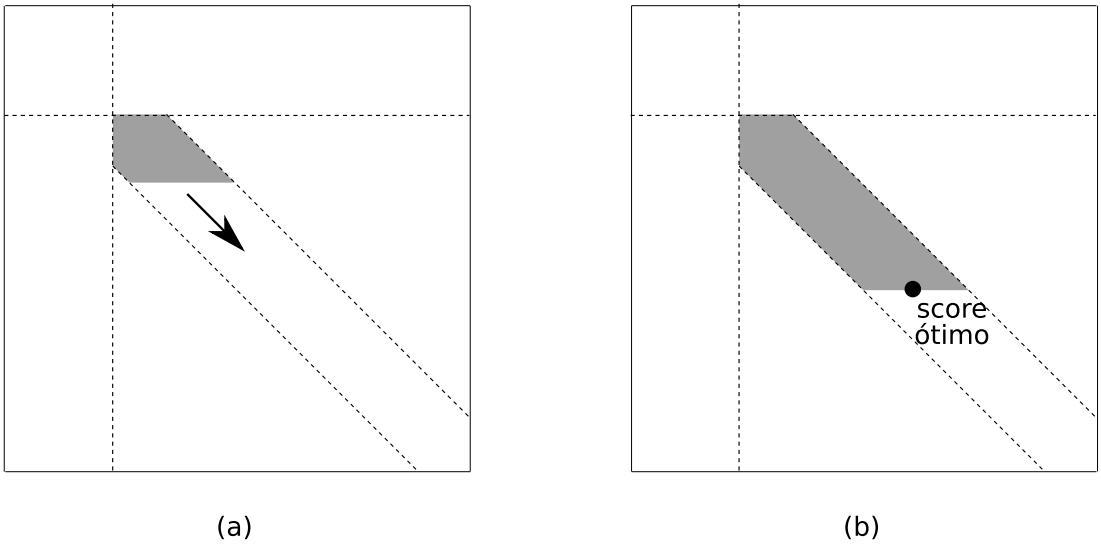


Figura 5.8: Alocação de memória. (a) A área cinza representa a região de memória que já foi alocada e teve seus valores calculados. (b) Encontrado um valor igual ao score ótimo procurado. A partir deste ponto inicia-se o *traceback* e construção do alinhamento com base nos valores calculados.

que indicam uma faixa de área bem reduzida para a recuperação de um alinhamento e, alocando apenas a memória necessária para processar a faixa até o ponto onde se localiza o fim do alinhamento, dependendo do valor da divergência e do comprimento do alinhamento, o processador pode não ter disponível a quantidade de memória suficiente para armazenar o trecho processado da faixa.

A Figura 5.9 ilustra como a obtenção dos alinhamentos é feita utilizando-se uma quantidade restrita de memória. Na Figura 5.9a, o algoritmo modificado de Fickett se iniciou e vai alocando memória à medida que o processamento avança, até o ponto em que a memória alocada atinge um limite estipulado, mas ainda não foi encontrado um valor igual ao *score* procurado. Uma das linhas da matriz é então mantida em memória, e o resto é liberado. A linha mantida em memória é chamada de *linha-cache*. Na Figura 5.9b vê-se a memória sendo novamente alocada e o processamento continua. Toda vez que o limite de memória é atingido, uma linha-cache é definida e a área de memória da região processada é liberada. Na Figura 5.9c foi encontrado o valor correspondente ao *score* desejado e são mostradas também as linhas-cache criadas.

Na Figura 5.9d, o procedimento de *traceback* é iniciado e procede até a linha-cache mais próxima. Para que o alinhamento possa continuar, é necessário alocar e reprocessar a região que se compreende entre a linha-cache atual e a linha-cache imediatamente superior. É mostrada na Figura 5.9e a região abaixo da linha-cache atual sendo liberada da memória, e o alinhamento continuando até a próxima linha-cache. Linhas-cache que não são mais necessárias também são removidas da memória. Na Figura 5.9g, o algoritmo modificado de Fickett chega ao fim tendo produzido o alinhamento desejado e mantendo o uso de memória dentro do limite estipulado.

O uso de linhas-cache permite produzir os alinhamentos em um espaço restrito de memória. O reprocessamento envolvido pode chegar a dobrar o tempo necessário para a obtenção de um alinhamento. No entanto, na maioria dos alinhamentos locais ótimos obtidos, os valores de divergência são pequenos o suficiente para que não seja necessário o uso de linhas-cache. Assim, não ocorre reprocessamento dos valores de similaridade, sendo o alinhamento obtido com apenas um passo de processamento e *traceback*.

Na Figura 5.10 é apresentada uma visão geral do ***z***Align. A primeira fase é omitida propositadamente.

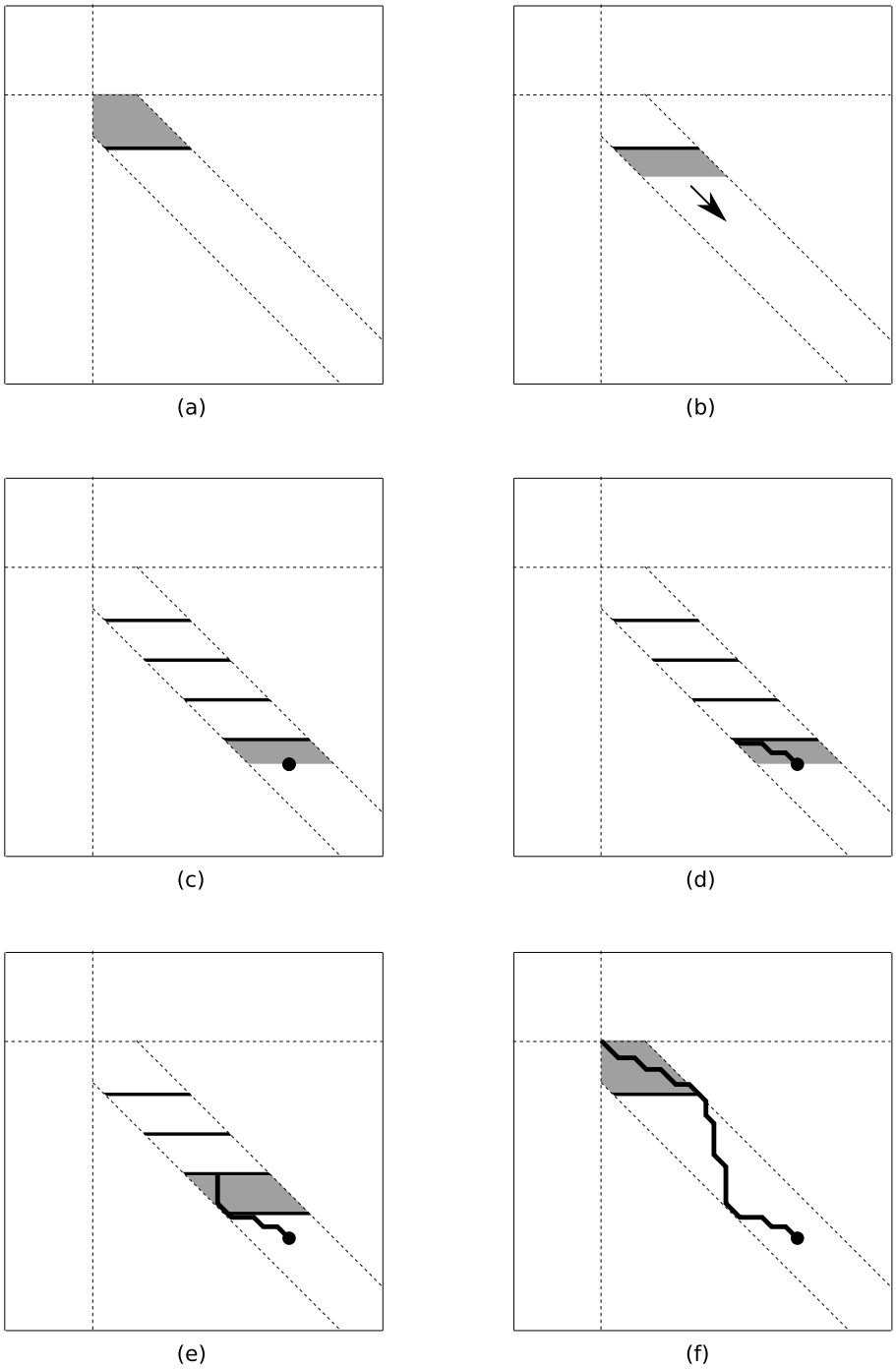


Figura 5.9: Linhas-cache. Regiões em cinza indicam a área de memória alocada. (a) Início do processamento, limite de memória atingido e linha-cache definida. (b) Apenas a linha-cache é guardada e o processamento continua. (c) Linhas-cache definidas e *score* ótimo encontrado. (d) Início do *traceback*. (e) Reprocessamento a partir das linhas-cache para continuação do *traceback*. (f) Alinhamento recuperado.

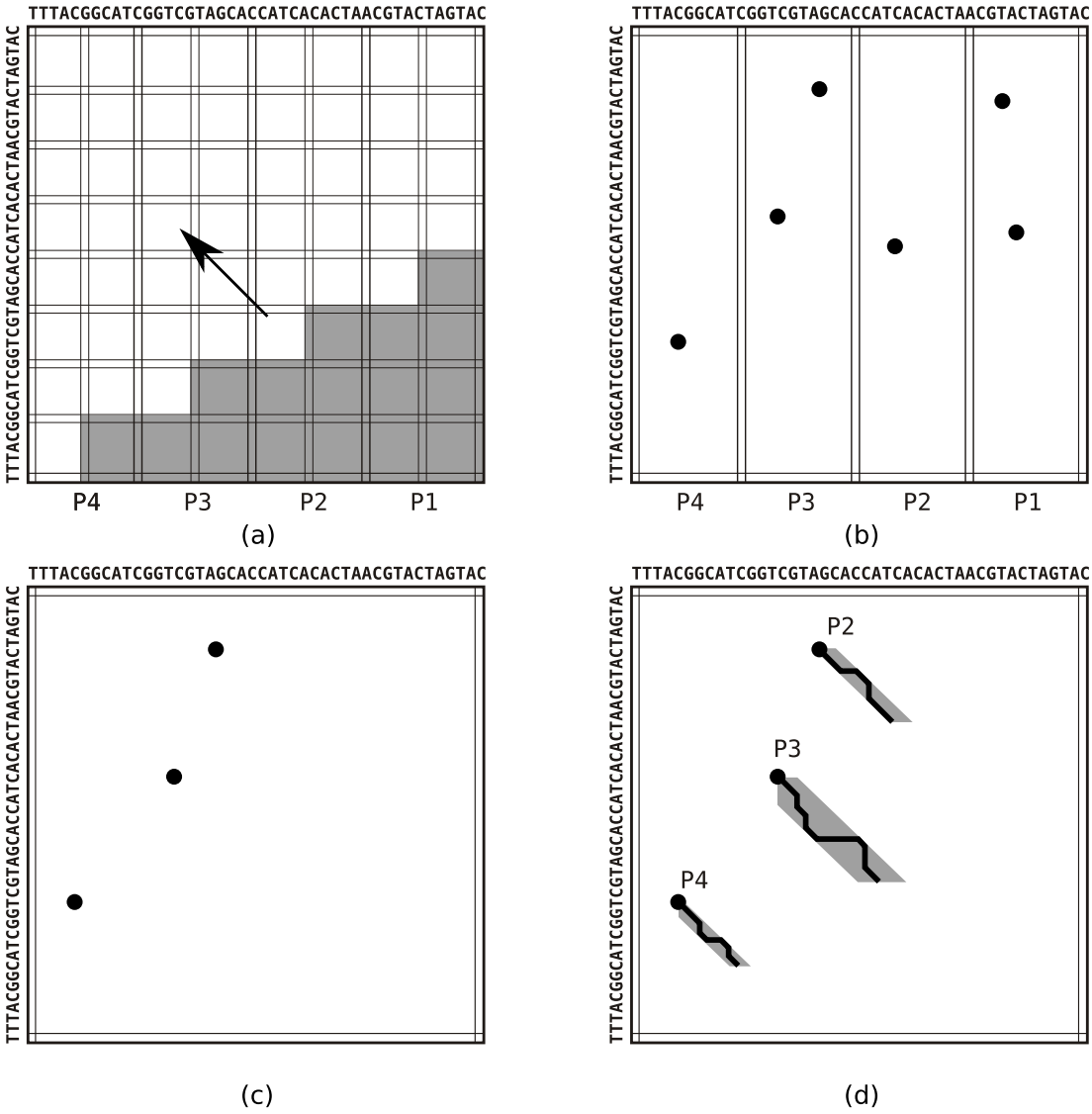


Figura 5.10: Visão geral do ***z***Align. (a) Fase de programação dinâmica. A matriz $S_{i,j}$ é processada em onda sobre S^{inv} e T^{inv}. (b) Final da fase de programação dinâmica, mostrando a posição final dos melhores alinhamentos encontrados. (c) Na fase de agregação, permanecem apenas os alinhamentos de *score* ótimo. (d) Fase de alinhamento, mostrando a distribuição de trabalho, as regiões processadas e os alinhamentos produzidos.

Capítulo 6

Resultados experimentais

O *z*Align foi implementado usando C++ e MPI. Testes foram realizados para verificar o seu desempenho e a precisão dos resultados produzidos. A seguir, serão descritos o ambiente onde foram conduzidos os testes, os dados utilizados, os procedimentos adotados e os resultados obtidos.

6.1 Ambiente de implementação e testes

A implementação do *z*Align foi feita em linguagem C++, utilizando o compilador GCC 3.4.3, no sistema operacional GNU/Linux kernel 2.4.27 em uma distribuição Debian 3.1 r1.

Para a comunicação entre os processos foi utilizada o padrão MPI, implementado pela biblioteca MPICH 1.2.7. Para monitorar o desempenho da comunicação dos processos através do MPI, foi utilizada a biblioteca mpiP 2.8.2.

Os testes foram executados sobre um *cluster* dedicado de 8 computadores, cada um com 2 processadores AMD Athlon MP 1900+ e 1 GB de memória RAM. Os computadores são interligados por um Switch Gigabit Ethernet.

As subdivisões horizontais (h) e verticais (v) utilizadas na obtenção dos resultados e os tamanhos das seqüências comparadas são descritas na Tabela 6.1.

Para a quarta fase do *z*Align, a utilização de memória por processador para a obtenção dos alinhamentos ficou limitada a 64 MB.

	Subdivisões	
Seqüências	**Horizontais**	**Verticais**
1 Kbp	1	10
10 Kbp	1	102
50 Kbp	5	573
150 Kbp	13	1.600
500 Kbp	60	5.000
1 Mbp	85	10.400
3 Mbp	273	31.000

Tabela 6.1: Subdivisões horizontais e verticais para os testes realizados.

6.2 Seqüências analisadas

Para verificar se a implementação de *z*Align está correta, bem como para obter os resultados de desempenho, foram utilizadas seqüências genéticas reais, obtidas a partir da página do NCBI (*National Center for Biotechnology Information*) no endereço http://www.ncbi.nlm.nih.gov/.

Foram escolhidas seqüências cujos tamanhos aproximados são de 1 Kbp, 10 Kbp, 50 Kbp, 150 Kbp, 500 Kbp, 1 Mbp e 3 Mbp. A implementação trabalha com seqüências de DNA no formato FASTA. A Tabela 6.2 mostra as seqüências utilizadas. Estão presentes na tabela os nomes dos organismos cujas seqüências foram alinhadas, o tipo de seqüência utilizada, o nº de acesso RefSeq a partir do qual a seqüência pode ser recuperada na base do NCBI, o comprimento em pares de bases e o comprimento aproximado.

Deste ponto em diante, as seqüências serão referenciadas pelos tamanhos aproximados das mesmas.

Tamanho		**Organismo**	**Tipo**	**Nº de Acesso**
Aprox.	**Real**			
1 Kbp	1.440bp	*Acetobacter pasteurianus* plasmid pAP12875 [17]	Plasmídio	NC_004991
	2.747bp	*Bacteroides fragilis* plasmid pBI143 [48]	Plasmídio	NC_005026
10 Kbp	10.035bp	HIV-1 isolate MB2059 from Kenya [40]	Genoma	AF133821
	10.280bp	HIV-1 isolate SF33 from USA [57]	Genoma	AY352275
50 Kbp	56.574bp	*Chaetosphaeridium globosum* mitochondrial DNA [53]	DNA Mitocondrial	AF494279
	57.473bp	*Allomyces macrogynus* [42]	DNA Mitocondrial	NC_001715
150 Kbp	162.114bp	*Human herpesvirus* 6B [13]	Genoma	NC_000898
	171.823bp	*Human herpesvirus* 4 [11]	Genoma	NC_007605
500 Kbp	542.869bp	*Agrobacterium tumefaciens* str. C58 plasmid AT [21]	Plasmídio	NC_003064
	536.165bp	*Rhizobium sp.* plasmid pNGR234a [19]	Plasmídio	NC_000914
1 Mbp	1.044.459bp	*Chlamydia trachomatis* A/HAR-13 [8]	Genoma	CP000051
	1.072.950bp	*Chlamydia muridarum* Nigg[51]	Genoma	AE002160
3 Mbp	3.147.090bp	*Corynebacterium efficiens* YS-314 DNA[41]	Genoma	BA000035
	3.282.708bp	*Corynebacterium glutamicum* ATCC 13032[29]	Genoma	BX927147

Tabela 6.2: Seqüências analisadas

6.3 Medidas de desempenho paralelo

6.3.1 *Speedups* relativos

A implementação do *z*Align foi testada para as seqüências de aproximadamente 1 Kbp, 10 Kbp, 50 Kbp, 150 Kbp, 500 Kbp, 1 Mbp e 3 Mbp descritas na Tabela 6.2. Para as seqüências de 1 Kbp até 500 Kbp, o *z*Align foi testado utilizando-se 1, 2, 4, 8 e 16 processadores. Já as seqüências de 1 Mbp e 3 Mbp foram executadas somente utilizando-se 16 processadores.

Os tempos de execução são mostrados na Tabela 6.3. Os valores em questão refletem o tempo total de execução medido utilizando-se chamadas à função MPI_Wtime logo após MPI_Init e logo antes de MPI_Finalize.

Vale ressaltar que o tempo de execução para 1 processador também foi obtido a partir da execução da implementação paralela, logo temos *speedups* relativos[18] referentes às várias execuções da implementação. Estes valores de *speedup* são mostrados na Tabela 6.4 e um gráfico resultante destes valores é mostrado na Figura 6.1.

Percebe-se que para 1 Kbp e 10 Kbp, o desempenho do *z*Align é insatisfatório. Isto pode ser explicado pelo fato de o tempo gasto com comunicação e sincronização exceder o tempo de processamento da matriz de similaridades e dos alinhamentos. Esse aspecto será abordado em detalhes na Seção 6.3.2.

Já para os alinhamentos acima de 50 Kbp, o *z*Align tem um bom desempenho, apresentando diversos *speedups* superlineares. Este comportamento é explicado pela divisão das linhas da matriz de similaridades entre os processadores. Quanto maior o número de processadores, menor se torna a largura dos blocos com que cada processador trabalha. Isto diminui significativamente o comprimento das linhas processadas, levando provavelmente a um melhor aproveitamento das linhas de *cache* do processador.

O melhor *speedup* observado foi para as seqüências de 150 Kbp, apresentando o melhor equilíbrio entre tempo de processamento e tempo de comunicação. As seqüências de 500 Kbp produziram bom desempenho, apesar de um pouco inferior ao observado nas seqüências de 150 Kbp. Isso leva a crer que, para seqüências de comprimento acima de 500 Kbp, o tempo de comunicação começa ter um impacto no *speedup* obtido.

O tempo de alinhamento para as seqüências de 1 Mbp foi de aproximadamente 1 hora e 43 minutos utilizando-se 16 processadores. Supondo-se que, para estas mesmas seqüências o desempenho do *z*Align ainda assim produza *speedup* linear, estima-se que o alinhamento destas mesmas seqüências em 1 processador levaria aproximadamente 1 dia, 3 horas e 32 minutos.

Usando a mesma extrapolação, para o caso das seqüências de 3 Mbp, cujo processamento demorou 13 horas e 31 minutos em 16 processadores, estima-se que em 1 processador seriam necessários, aproximadamente, 9 dias para produzir os mesmos resultados.

Tamanho		Número de processadores				
Seq.	**Alin.**	**1**	**2**	**4**	**8**	**16**
1 Kbp	10	0,28s	0,16s	0,13s	0,12s	0,18s
10 Kbp	9.111	8,42s	4,18s	2,34s	1,34s	0,95s
50 Kbp	80	254,96s	117,54s	59,90s	30,55s	16,69s
150 Kbp	18	2.161,69s	994,86s	500,39s	249,43s	127,97s
500 Kbp	96	18.041,40s	8.680,68s	4.344,82s	2.148,08s	1.093,38s
1 Mbp	471.621	-	-	-	-	6.197,04s
3 Mbp	14.537	-	-	-	-	48.704,70s

Tabela 6.3: Tempos de execução em segundos para o *z*Align e tamanho do maior alinhamento encontrado para seqüências de aproximadamente 1 Kbp, 10 Kbp, 50 Kbp, 150 Kbp, 500 Kbp, 1 Mbp e 3 Mbp.

Tamanho	Número de processadores				
	1	**2**	**4**	**8**	**16**
1 Kbp	1	1,76	2,12	2,47	1,61
10 Kbp	1	2,01	3,60	6,28	8,85
50 Kbp	1	2,17	4,26	8,35	15,27
150 Kbp	1	2,17	4,32	8,67	16,89
500 Kbp	1	2,08	4,15	8,40	16,50

Tabela 6.4: Speedups relativos do *z*Align para as execuções com 1, 2, 4, 8, e 16 processadores com as seqüências de aproximadamente 1 Kbp, 10 Kbp, 50 Kbp, 150 Kbp e 500 Kbp.

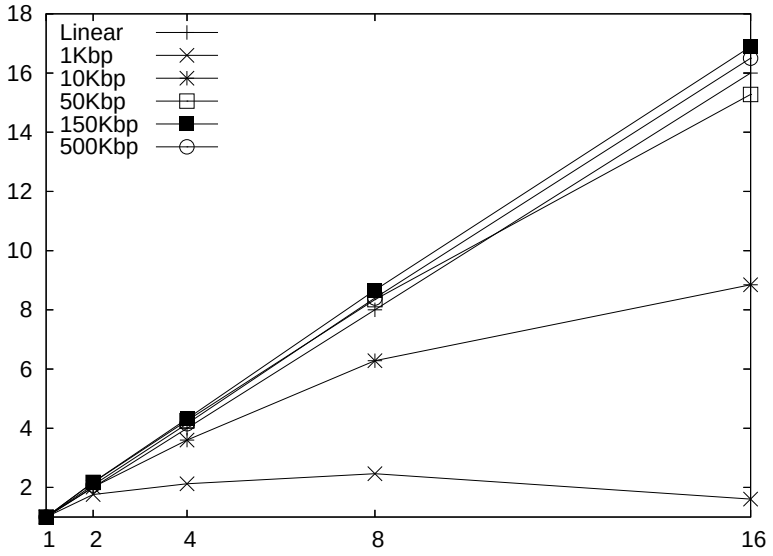


Figura 6.1: *Speedups* relativos do *z*Align para as execuções com 1, 2, 4, 8, e 16 processadores com as seqüências de aproximadamente 1 Kbp, 10 Kbp, 50 Kbp, 150 Kbp e 500 Kbp.

6.3.2 Tempos de comunicação MPI

Os tempos gastos com a comunicação usando MPI também foram mensurados e são apresentados graficamente nesta seção. Para obter tais dados, foram utilizados os relatórios gerados pela biblioteca mpiP (*Lightweight, Scalable MPI Profiling*) obtida no endereço http://www.llnl.gov/CASC/mpip/.

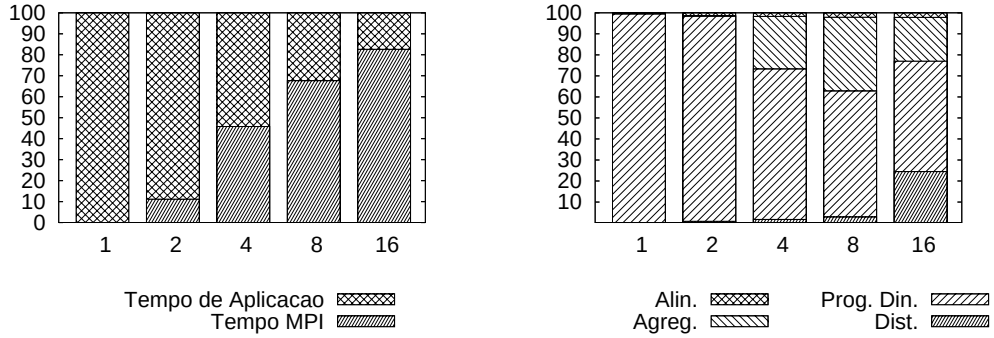
Nas Figuras 6.2a, a 6.8a são mostradas as razões entre o tempo de processamento e o tempo consumido com a comunicação MPI para 1, 2, 4, 8, e 16 processadores. Aqui, **Tempo de MPI** representa a fração do tempo total de processamento onde a execução encontra-se dentro de alguma função MPI. O tempo ocioso de espera de uma função MPI_recv por uma mensagem e o tempo dentro de uma função MPI_barrier contam como **Tempo MPI**, por exemplo. O resto do tempo é representado por **Tempo de Aplicação**.

Nas Figuras 6.2b, a 6.8b (quando se aplicar), são mostradas as frações dos tempos gastos com cada fase do *z*Align. Aqui, **Dist.** equivale ao tempo gasto na primeira fase, onde são distribuídas as seqüências entre os processadores, **Prog. Din.** equivale ao tempo gasto na segunda fase onde são calculadas as matrizes de programação dinâmica, **Agreg.** representa o tempo gasto para os processadores enviarem ao processador 1 os dados coletados e **Alin.** representa o tempo gasto para produzir os alinhamentos encontrados. Vale ressaltar que nestas figuras, o tempo de comunicação está disperso dentro das fases, sendo que a maioria do tempo de comunicação está na segunda fase.

6.3.2.1 Análise das seqüências de 1 Kbp

Na Figura 6.2 vê-se que o pequeno tamanho da seqüência não provê trabalho suficiente. O tempo de execução é então dominado pelos custos de comunicação da seqüência entre os processadores, chegando a 82% do tempo total de execução em 16 processadores. Isto justifica os *speedups* ruins obtidos para estas seqüências.

Percebe-se também que, para 16 processadores, o desempenho possui um comportamento desviado do padrão, o que pode ser justificado pelo fato de que uma vez tendo um tempo de execução muito curto, por volta de 0,003 segundos, até mesmo processos do sistema operacional que possam estar em execução em um dos nodos podem afetar o desempenho em algum ponto do *z*Align.



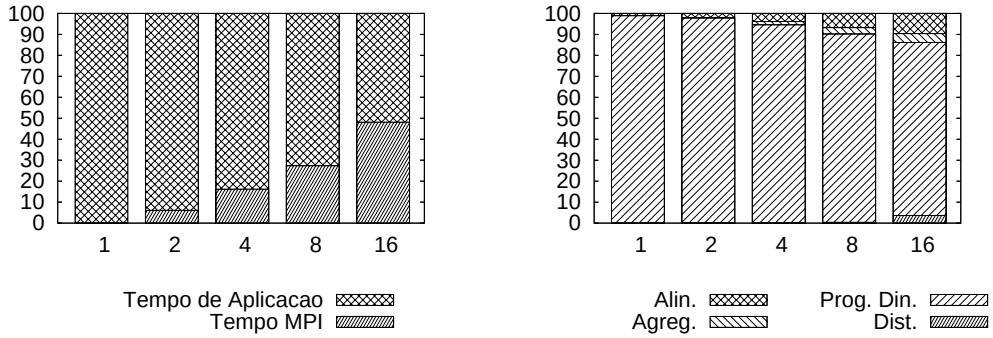
(a) Tempo de Comunicação (b) Fases do *z*Align

Figura 6.2: *Breakdown* da execução para seqüências de 1 Kbp.

6.3.2.2 Análise das seqüências de 10 Kbp

Na Figura 6.3, percebe-se que o tempo consumido com a comunicação MPI ainda é excessivo quando comparado com o tempo total de execução. A divisão de tempo entre as fases já se apresenta com um comportamente mais próximo ao do esperado, com o tempo necessário para o cálculo da matriz de similaridades dominando o tempo de processamento.

Os tempos de distribuição das seqüências, de agregação dos resultados e de alinhamento passam a ter uma participação maior à medida que mais processadores vão sendo adicionados. No entanto, tais tempos são os mesmos para as várias execuções do *z*Align. O que ocorre é que o tempo consumido pela segunda fase diminui, fazendo com que os outros tempos tenham uma participação proporcionalmente maior no tempo total de execução.

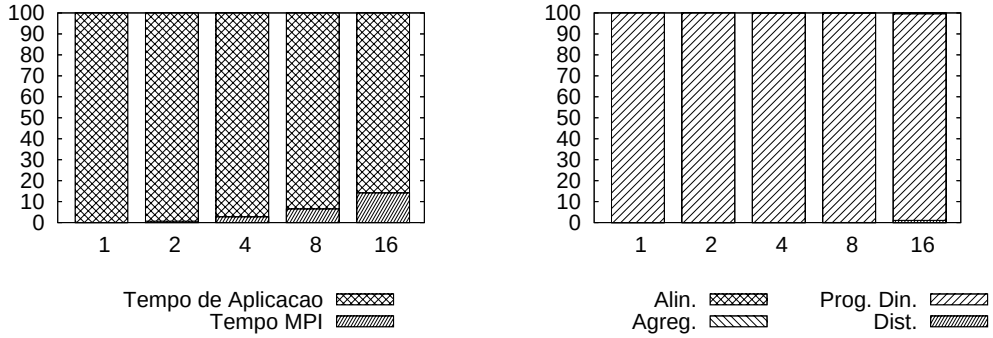


(a) Tempo de Comunicação (b) Fases do *z*Align

Figura 6.3: *Breakdown* da execução para seqüências de 10 Kbp.

6.3.2.3 Análise das seqüências de 50 Kbp

A Figura 6.4 mostra que com seqüências a partir de 50 Kbp, o tempo de execução é praticamente todo em função da segunda fase. Nota-se que a fração do tempo consumido com comunicação MPI continua a decair.

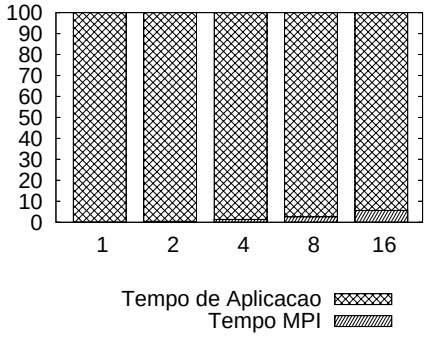


(a) Tempo de Comunicação (b) Fases do *z*Align

Figura 6.4: *Breakdown* da execução para seqüências de 50 Kbp.

6.3.2.4 Análise das seqüências de 150 Kbp

A Figura 6.5 mostra que o tempo de processamento passa a ter ainda mais contribuição no tempo de execução do que o tempo de comunicação. A divisão do tempo pelas fases do *z*Align foi propositadamente omitida pois o custo da segunda fase contribui com mais de 99,5% do tempo total de execução, mesmo no caso de execução em 16 processadores.

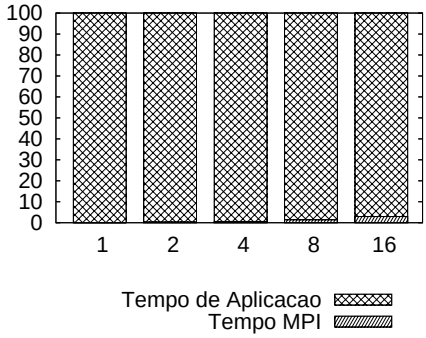


(a) Tempo de Comunicação

Figura 6.5: *Breakdown* da execução para seqüências de 150 Kbp.

6.3.2.5 Análise das seqüências de 500 Kbp

Na figura 6.6, as mesmas observações feitas para a execução com seqüências de 150 Kbp continuam valendo, tendo sido omitido novamente o gráfico com a divisão do tempo pelas fases do *z*Align.



(a) Tempo de Comunicação

Figura 6.6: *Breakdown* da execução para seqüências de 500 Kbp.

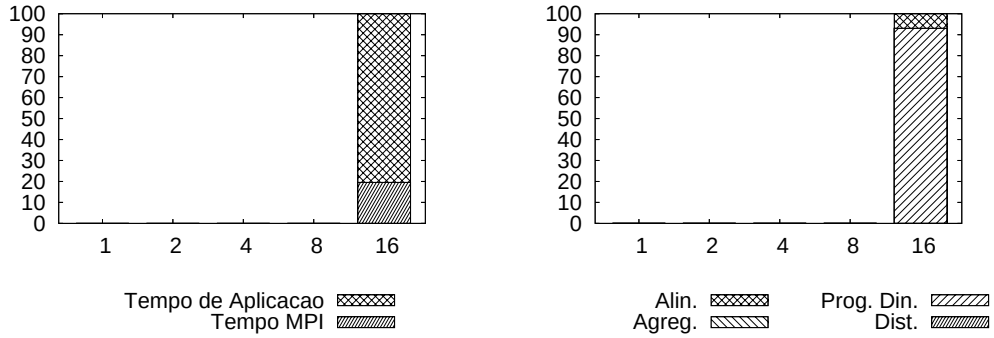
6.3.2.6 Análise das seqüências de 1 Mbp

Para o caso de seqüências de 1 Mbp, um comportamento diferenciado é mostrado na Figura 6.7. A fase de alinhamento para estas seqüências demora 431,75 segundos. Dentre as seqüências analisadas, este é o único caso em que a fase de alinhamento tem um custo maior que 2 segundos. O tempo gasto com a produção

do alinhamento é de cerca de 7% do tempo total de processamento, o que pode ser visto na Figura 6.7b.

A Figura 6.7a mostra um tempo MPI maior do que o normal, chegando a 19,54% do tempo total de execução. O que acontece realmente é que, na fase de alinhamento, apenas 1 dos processadores encontra-se ocupado enquanto os outros 15 processadores encontram-se aguardando em uma chamada MPI_recv. Como o tempo medido pela biblioteca mpiP registra na verdade o tempo gasto dentro das funções MPI e não o tempo efetivamente gasto com a comunicação de dados, o tempo MPI registrado inclui também todo o tempo decorrido entre a chamada da função MPI_recv e o retorno desta mesma função.

As Figuras 6.7a e 6.7b não mostram os tempos de execução para 1, 2, 4 e 8 processadores pois estes tempos não foram medidos devido ao comprimento das seqüências.



(a) Tempo de Comunicação (b) Fases do *z*Align

Figura 6.7: *Breakdown* da execução para seqüências de 1 Mbp.

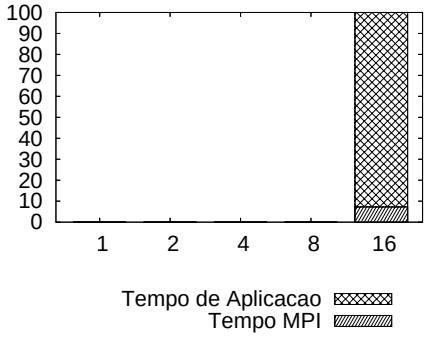
6.3.2.7 Análise das seqüências de 3 Mbp

Na execução com seqüências de 3 Mbp, o tempo de execução volta a ser novamente determinado praticamente apenas pela segunda fase, sendo assim o gráfico de fases foi novamente omitido.

Vê-se, no entanto, na Figura 6.8 que o tempo gasto com a comunicação MPI ocupa uma parcela maior do tempo total. Para este caso, isto se deve ao excessivo número de vezes que a função MPI_send é utilizada. Para estas seqüências, esta função é chamada 2.325.000 vezes durante a segunda fase, o que dá uma média de 2,98 chamadas por segundo por processador, sendo que em cada chamada são transferidos 410 *bytes* apenas.

Isto leva a crer que o valor de v utilizado para subdividir verticalmente a matriz de similaridades está causando um excesso de comunicação entre os processadores. Diminuir o valor utilizado provavelmente diminuirá a quantidade de chamadas à função MPI_send, bem como aumentará a quantidade de *bytes* transferidos a cada chamada, utilizando de forma mais eficiente a comunicação MPI, diminuindo provavelmente o tempo de execução do *z*Align.

Também devido ao comprimento destas seqüências, não foram medidos os tempos de execução para 1, 2, 4 e 8 processadores, estando estes ausentes na Figuras 6.8.



(a) Tempo de Comunicação

Figura 6.8: *Breakdown* da execução para seqüências de 3 Mbp.

6.4 Reprocessamento na fase de alinhamento

A última fase do *z*Align foi analisada para avaliar o custo de se produzir os alinhamentos a partir dos dados produzidos na segunda fase. A quantidade de reprocessamento para um alinhamento é obtida da seguinte forma: Supondo que um alinhamento local $\mathcal{A}$ esteja definido sobre as subseqüências $S[i' \ldots i'']$ e $T[j' \ldots j'']$. Sabendo-se que um alinhamento global entre estas subseqüências produz um alinhamento idêntico ao alinhamento $\mathcal{A}$, toma-se $R_g = (i'' - i') \times (j'' - j')$ a área da matriz de similaridades necessária para a produção deste mesmo alinhamento global. Se, para este mesmo alinhamento global, tomarmos como R_{Fm} a área realmente processada pelo algoritmo modificado de Fickett na quarta fase, temos que o reprocessamento R é a razão entre estas duas medidas, ou seja, $R = \frac{R_{Fm}}{R_g}$

Na Tabela 6.5 são mostrado os dados obtidos para a quarta fase. São mostrados as divergências máximas, o comprimento, a quantidade de reprocessamento e o tempo para a produção dos alinhamentos na quarta fase. As divergências superior e inferior, juntamente com o comprimento do alinhamento, são o fator determinante do tempo necessário para se produzir os alinhamentos.

O uso dos valores de divergência e as adaptações do algoritmo de Fickett permitiram uma economia muito grande de reprocessamento. No caso do alinhamento obtido para as seqüências de 1 Mbp, o alinhamento compreende-se numa região de $463.699bp \times 466.054bp$. Como o *z*Align está utilizando 4 *bytes* para representar cada valore de $S_{i,j}$, $P_{i,j}$, $Q_{i,j}$ e 1 *byte* para os valores de $e_{i,j}$, o espaço exigido para produzir este alinhamento utilizando o algoritmo de Gotoh gira em torno de $(463699 \times 466054 \times (4 + 4 + 4 + 1))$ *bytes*, o que se traduz em aproximadamente 2,55 TB de memória.

Com o algoritmo de Fickett e os valores de divergência, isso cairia para $((99 + 4027 + 1) \times 466054 \times 13)$ *bytes*, ou aproximadamente 23,28 GB. Com o uso das linhas-cache o alinhamento foi produzido dentro de um espaço de aproximadamente 64 MB de memória em $431,769s$ tendo sido reprocessada apenas $1,78\%$[1] da matriz referente à região que compreende o alinhamento.

[1]Das seqüências analisadas, apenas a seqüência de 1 Mbp exigiu 2 passos para a produção do alinhamento. Caso não fosse necessário o 2º passo, apenas uma região equivalente a $0,89\%$ da sub-matriz correspondente à região de similaridade teria de ser reprocessada.

	Divergência				
Seq.	**Sup.**	**Inf.**	**Comprim.**	**Reprocess.**	**Tempo**
1 Kbp	0	0	10	10,00%	0,003s
10 Kbp	-4	32	9.111	0,40%	0,900s
50 Kbp	-1	0	80	2,56%	0,034s
150 Kbp	0	0	18	2,77%	0,039s
500 Kbp	0	0	96	1,05%	0,040s
1 Mbp	-99	4.027	471.621	1,78%	431,759s
3 Mbp	-40	43	14.537	0,59%	0,207s

Tabela 6.5: Divergência máxima e reprocessamento efetuado para os alinhamentos obtidos na fase de alinhamento para as seqüências de 1 Kbp, 10 Kbp, 50 Kbp, 150 Kbp, 500 Kbp, 1 Mbp e 3 Mbp.

6.5 Comparações com o BLAST

Foram realizadas comparações com a implementação do BLAST disponível no NCBI através do endereço http://www.ncbi.nih.gov/BLAST/. Foi utilizada a versão 2.2.13 do BLAST para o sistema operacional Linux. Foi utilizado o programa **bl2seq** que faz a comparação entre duas seqüências biológicas. O algoritmo usado pelo **bl2seq** foi o **blastn**, com *gaps* habilitados e busca apenas na fita superior (*top strand*).

O uso do BLAST como parâmetro de comparação justifica-se pelo fato deste ser amplamente usado pelos biólogos em suas análises. Com isso, pretende-se comparar nesta seção as características dos alinhamentos produzidos pelo ***z***Align e pelo BLAST.

Tanto no Blast quanto no ***z***Align, foram utilizadas as seguintes pontuações: 1 para *match*, -3 para *mismatch*, -5 para abertura de *gap* e -2 para extensão de *gap*.

	Alinhamentos produzidos			
	Quantidade		Comprimento Máximo	
Seqüência	***z*Align**	**BLAST**	***z*Align**	**BLAST**
1 Kbp	5	-	10	-
10 Kbp	1	38	9.111	2.554
50 Kbp	1	101	80	79
150 Kbp	1	9	18	18
500 Kbp	2	19	96	92
1 Mbp	1	1.593	471.621	10.763
3 Mbp	1	5.057	14.537	5.329

Tabela 6.6: Quantidade e comprimento máximo de alinhamentos encontrados pelo ***z***Align em comparação com o BLAST

A Tabela 6.6 mostra um resumo das quantidades e comprimentos dos alinhamentos encontrados tanto pelo ***z***Align quanto pelo BLAST.

Como o ***z***Align procura apenas pelos alinhamentos que possuem *score* ótimo, a

quantidade de alinhamentos produzidos pelo mesmo é bem inferior à quantidade de alinhamentos produzidos pelo BLAST. Em compensação, os tamanhos dos alinhamentos produzidos pelo *z*Align são quase sempre superiores ou, pelo menos, do mesmo tamanho. No caso da seqüência de 1 Mbp, o alinhamento produzido chega a ser 43 vezes maior que o melhor alinhamento encontrado para o BLAST, para as mesmas seqüências.

A Figura 6.9 mostra exemplos de saída da implementação do *z*Align e do BLAST para seqüências de 50 Kbp.

```
--------------------------------------------------------------------------------
                                    zAlign
--------------------------------------------------------------------------------
Sequence s: 56574
file: samples/gi-22416996.fasta

Sequence t: 57473
file: samples/gi-11467573.fasta

Aligning: Start=(41274,33360) End=(56574,57473) Score=52 Divergence=(1,0)
Score =  52
Identities = 74/80 (92%), Gaps = 2/80 (2%)
Start = (41274,33360) End = (41352,33438)
Query 41274 AGACCGTCGGTTGTGTGTACAATCGCTACAGACTGGTTCATC-GGTGGGTGCCTGAAATG  41332
            ||||||||||||||.|.|||.||||||||||||||||||| || ||||||||||||||||
Sbjct 33360 AGACCGTCGGTTGTTTATACGATCGCTACAGACTGGTTC-TCTGGTGGGTGCCTGAAATG  33418

Query 41333 GTGCTTAATGTACAGTCGGA  41352
            ||||||||||.||||||||
Sbjct 33419 GTGCTTAATGTTCAGTCGGA  33438

--------------------------------------------------------------------------------
                                     BLAST
--------------------------------------------------------------------------------
Query= gi|22416996|gb|AF494279.1| Chaetosphaeridium globosum
mitochondrial DNA, complete  genome
         (56,574 letters)

>gi|11467573|ref|NC_001715.1| Allomyces macrogynus mitochondrion,
             complete genome
          Length = 57473

 Score =  101 bits (51), Expect = 8e-22
 Identities = 72/79 (91%)
 Strand = Plus / Plus


Query: 41274 agaccgtcggttgtgtgtacaatcgctacagactggttcatcggtgggtgcctgaaatgg 41333
            ||||||||||||||| | ||| ||||||||||||||||||   ||||||||||||||||||
Sbjct: 33360 agaccgtcggttgtttatacgatcgctacagactggttctctggtgggtgcctgaaatgg 33419


Query: 41334 tgcttaatgtacagtcgga 41352
            |||||||||| ||||||||
Sbjct: 33420 tgcttaatgttcagtcgga 33438
```

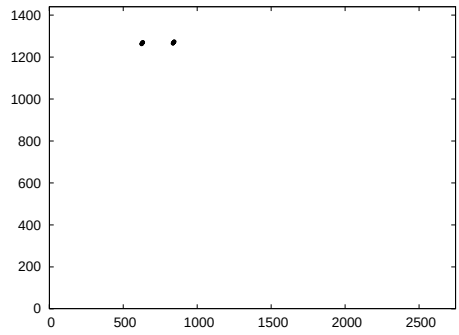
Figura 6.9: Exemplos de saída da implementação do *z*Align e do BLAST

A seguir são detalhados os alinhamentos obtidos tanto no *z*Align como no BLAST.

6.5.0.8 Análise das seqüências de 1 Kbp

Foram comparados plasmídios de *Acetobacter pasteurianus* e de *Bacteroides fragilis*. O BLAST não produziu nenhum alinhamento, pois os alinhamentos ótimos

existentes são curtos e devem estar abaixo de um limiar de corte do BLAST. Já o *z*Align produziu 5 alinhamentos, todos de comprimento 10. Os alinhamentos apresentam-se concentrados em duas regiões e a disposição deles é mostrada na Figura 6.10.



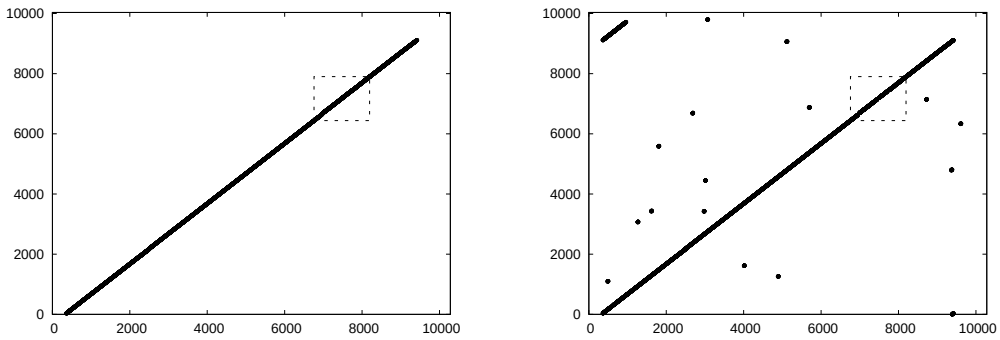
(a) *z*Align

Figura 6.10: Alinhamentos obtidos para seqüências de 1 Kbp (BLAST não produziu alinhamentos)

6.5.0.9 Análise das seqüências de 10 Kbp

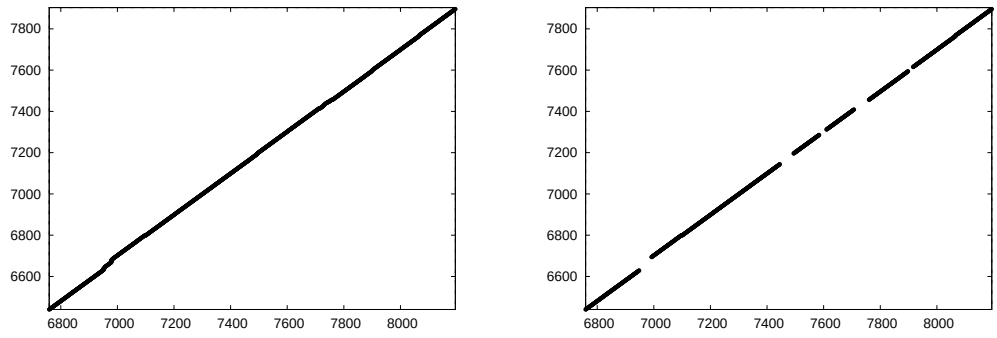
Foram analisadas seqüências de genomas de HIV-1 encontrados no Quênia e nos Estados Unidos. A Figura 6.11 mostra os alinhamentos obtidos pelo *z*Align e pelo BLAST para estas seqüências. O *z*Align encontrou 1 alinhamento de comprimento 9.111, enquanto que o BLAST encontrou um total de 38 alinhamentos, sendo que o maior comprimento é de 2.554.

A Figura 6.12 mostra com maior clareza de detalhes como os vários alinhamentos produzidos pelo BLAST se localizam próximos uns aos outros, fazendo parecer que para o BLAST, na Figura 6.11b, há apenas um grande alinhamento. Na verdade, o que ocorre é a presença de vários alinhamentos distintos, distantes entre si por algumas bases. O que o BLAST reporta como sendo vários alinhamentos, o *z*Align reporta como sendo apenas um alinhamento.



(a) *z*Align (b) BLAST

Figura 6.11: Alinhamentos obtidos para seqüências de 10 Kbp



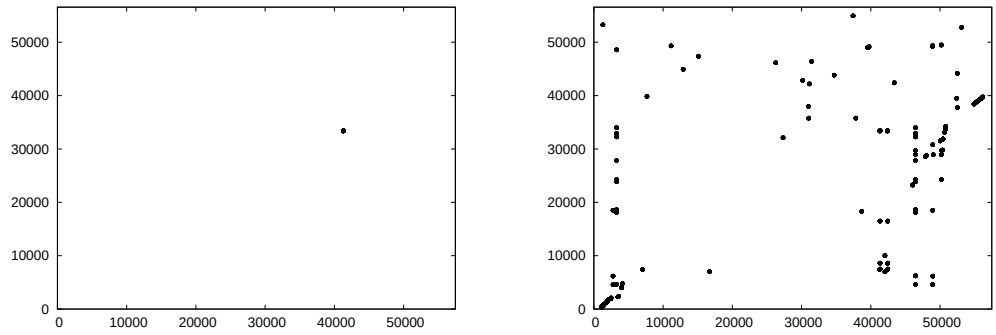
(a) *z*Align (b) BLAST

Figura 6.12: Detalhe dos alinhamentos obtidos para seqüências de 10K. A região destacada na Figura 6.11 é mostrada aqui em detalhes.

6.5.0.10 Análise das seqüências de 50 Kbp

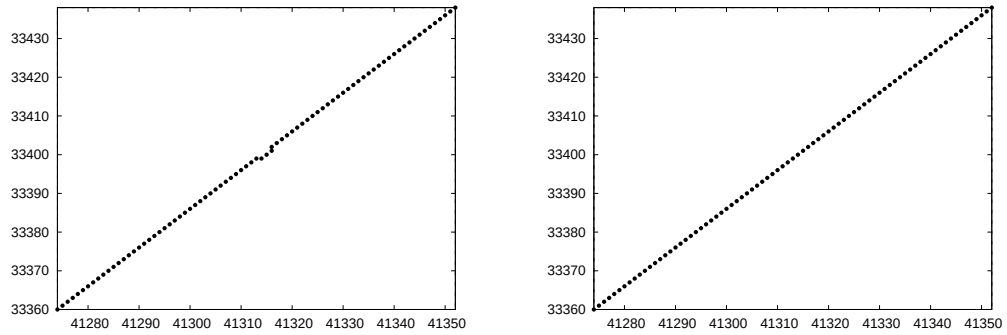
Aqui foi feita a comparação entre o DNA mitocondrial de *Chaetosphaeridium globosum* e de *Allomyces macrogynus*. O BLAST reporta 101 alinhamentos dispostos como mostra a Figura 6.13b, sendo que o maior deles é de comprimento 79. Na Figura 6.13a, é mostrada a localização do alinhamento ótimo, de comprimento 80.

A Figura 6.14 apresenta a região que compreende o alinhamento ótimo. Nota-se que o alinhamento produzido pelo BLAST é diferente do alinhamento ótimo produzido pelo *z*Align. Neste caso, o BLAST não reconhece alguns *gaps* como parte do alinhamento ótimo.



(a) *z*Align (b) BLAST

Figura 6.13: Alinhamentos obtidos para seqüências de 50 Kbp

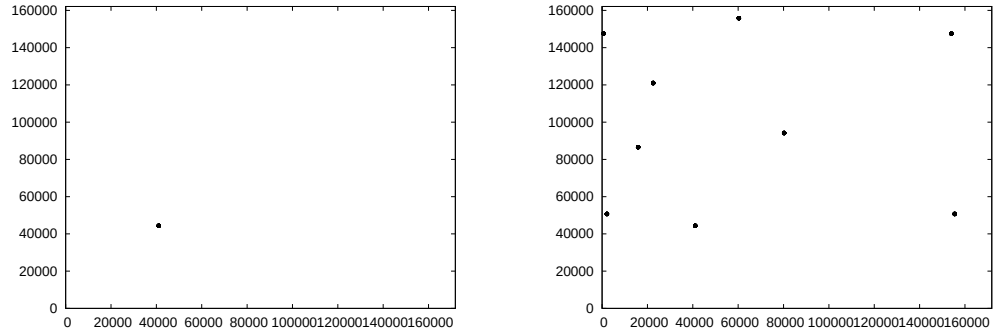


(a) *z*Align (b) BLAST

Figura 6.14: Detalhe dos alinhamentos obtidos para seqüências de 50K. É representada aqui a região correspondente ao alinhamento ótimo encontrado pelo *z*Align.

6.5.0.11 Análise das seqüências de 150 Kbp

Neste caso, os genomas de Herpesvírus Humano 6B e 4 foram comparados. Tanto o BLAST quanto o *z*Align produzem o alinhamento ótimo de comprimento 18. O BLAST produz ao total 9 alinhamentos, sendo apenas um deles o ótimo. A Figura 6.15 mostra os alinhamentos produzidos.

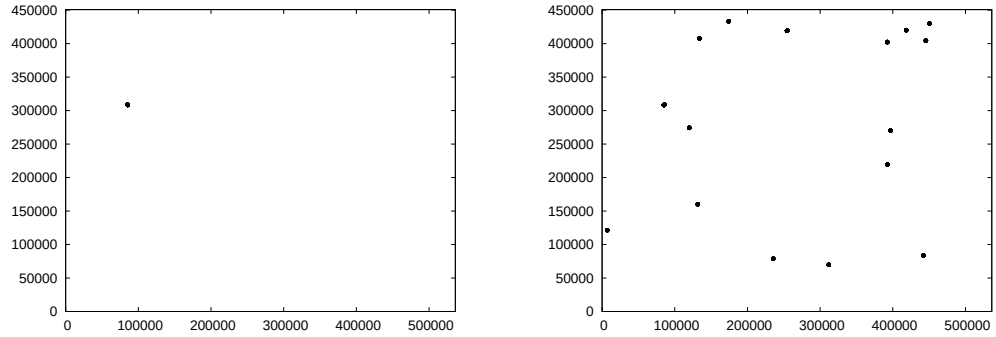


(a) *z*Align (b) BLAST

Figura 6.15: Alinhamentos obtidos para seqüências de 150 Kbp

6.5.0.12 Análise das seqüências de 500 Kbp

Foram comparados plasmídios de *Agrobacterium tumefaciens* e de *Rhizobium sp.* Aqui, o *z*Align novamente apresenta um alinhamento cujo comprimento é maior que o comprimento do maior alinhamento produzido pelo BLAST. O *z*Align apresenta 2 alinhamentos de comprimento 96, enquanto que o BLAST apresenta 19 alinhamentos cujo tamanho máximo chega a 92. Os alinhamentos ótimos encontrados pelo *z*Align encontram-se concentrados uma região. A Figura 6.16 mostra a disposição destes alinhamentos.



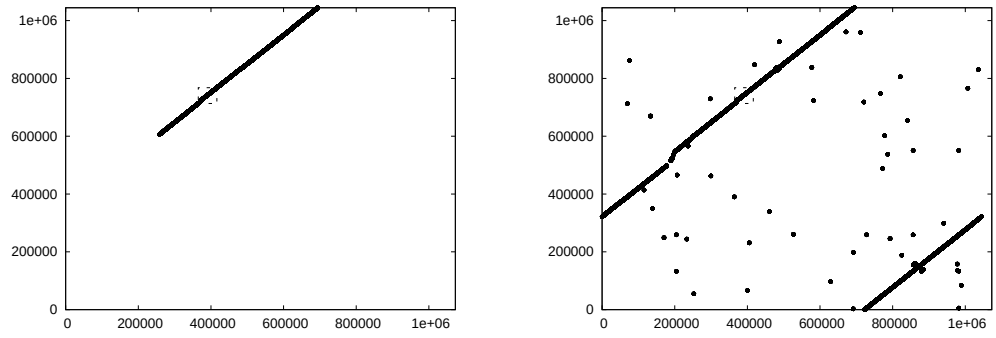
(a) ***z***Align (b) BLAST

Figura 6.16: Alinhamentos obtidos para seqüências de 500 Kbp

6.5.0.13 Análise das seqüências de 1 Mbp

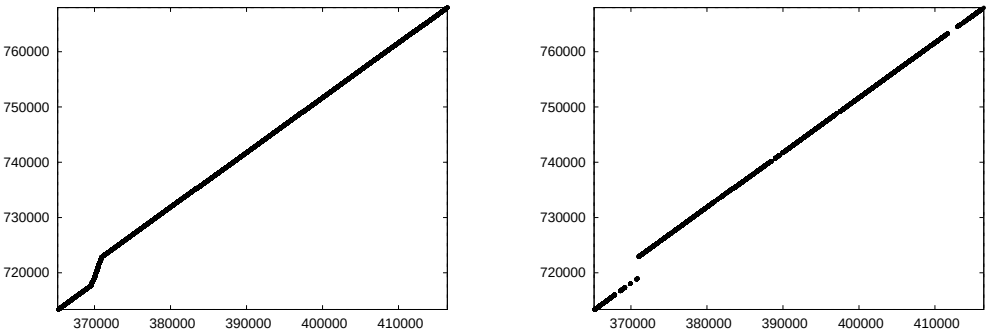
Para a comparação de seqüências de aproximadamente 1 Mbp foram utilizados os genomas de *Chlamydia trachomatis* e de *Chlamydia muridarum*. A Figura 6.17 mostra os alinhamentos produzidos pelo ***z***Align e pelo BLAST. Novamente os resultados do BLAST fazem parecer que este produziu um alinhamento de grande comprimento. A análise detalhada, na Figura 6.18, de um trecho correspondente ao alinhamento ótimo deixa clara a diferença existente entre a solução ótima e o resultado do BLAST.

O ***z***Align encontra 1 alinhamento de comprimento 471.621, enquanto que o BLAST produz 1.593 alinhamentos, sendo que o melhor deles possui comprimento 10.763.



(a) ***z***Align (b) BLAST

Figura 6.17: Alinhamentos obtidos para seqüências de 1 Mbp

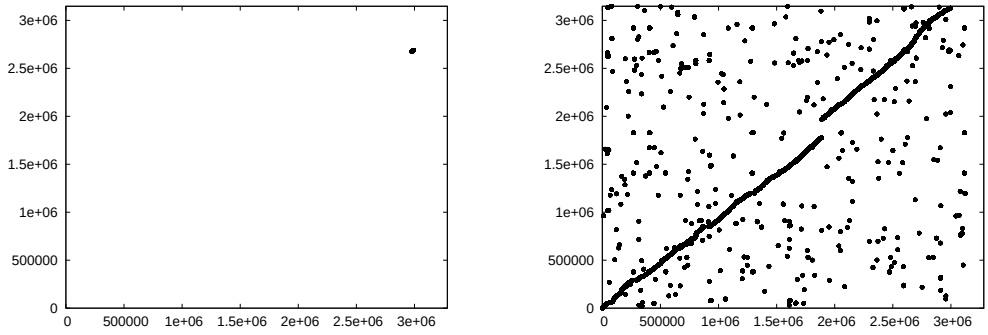


(a) *z*Align (b) BLAST

Figura 6.18: Detalhe dos alinhamentos obtidos para seqüências de 1 Mbp. A região destacada na Figura 6.17 é mostrada aqui em detalhes.

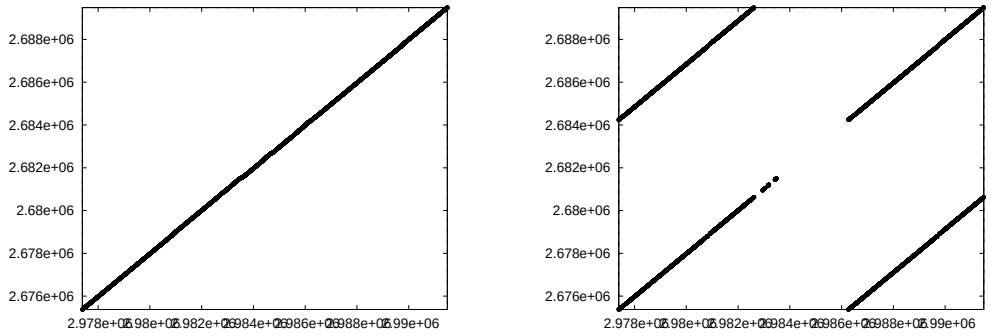
6.5.0.14 Análise das seqüências de 3 Mbp

Utilizando-se as seqüências de 3 Mbp, onde foram analisados os genomas de *Corynebacterium efficiens* e de *Corynebacterium glutamicum*, o BLAST produz 5.057 alinhamentos, sendo que o melhor deles possui comprimento 5.329. Na Figura 6.19 é mostrado novamente que o *z*Align produz 1 alinhamento ótimo de comprimento 14.537. Na Figura 6.20, um detalhe da região onde se situa o alinhamento ótimo ilustra mais um exemplo da diferença entre as duas soluções.



(a) *z*Align (b) BLAST

Figura 6.19: Alinhamentos obtidos para seqüências de 3 Mbp



(a) *z*Align (b) BLAST

Figura 6.20: Detalhe dos alinhamentos obtidos para seqüências de 3 Mbp. É representada aqui a região correspondente ao alinhamento ótimo encontrado pelo *z*Align.

6.6 Análise dos resultados obtidos

Os testes executados mostram que o *z*Align tem um desempenho muito bom para seqüências com tamanho superior a 50 Kbp. Para 150 Kbp e 500 Kbp, foram observados *speedups* relativos superlineares para 16 processadores.

Análises dos testes com 1 Mbp e 3 Mbp mostram que os parâmetros h e v de divisão vertical e horizontal devem ser estudados com cuidado a fim de encontrar um melhor equilíbrio entre o tamanho das seqüências e a quantidade de comunicação efetuada.

O *z*Align consegue lidar com a grande exigência de memória imposta pelos métodos exatos usados para a produção de alinhamentos entre seqüências grandes. O algoritmo de Gotoh, para as seqüências de 3 Mbp analisadas aqui, precisaria de no mínimo 9.39 TB de memória, considerando que cada entrada da matriz de similaridades fosse armazenada em 1 *byte* de memória. No *z*Align, a segunda fase utiliza espaço linear de memória, enquanto na quarta fase o usuário tem controle sobre a quantidade máxima de memória utilizada.

O reprocessamento efetuado na quarta fase para produzir um alinhamento tem um custo bem menor que custo de produzir este mesmo alinhamento com um alinhamento global das subseqüências associadas. O custo referente ao reprocessamento é ainda muito menor quando comparado com o trabalho realizado na segunda fase.

Os alinhamentos encontrados pelo *z*Align chegaram a ser até 43 vezes maiores que os encontrados pelo BLAST para as mesmas seqüências. Apenas para as seqüências de 150 Kbp, os alinhamentos ótimos encontrados possuem o mesmo tamanho. Isto mostra que o uso de heurísticas pode levar a informações imprecisas, não representando de maneira apropriada a correlação existente entre dois organismos.

A comparação com os resultados do BLAST mostrou que este, apesar de ser muito mais rápido que os métodos exatos baseados em programação dinâmica, produz alinhamentos de *score* às vezes muito inferiores aos ótimos produzidos pelos métodos exatos. Assim, o BLAST permite aos biólogos fazerem uma triagem

entre diversos organismos, mas a obtenção de informações precisas deve ser feita através de outras ferramentas. A execução do *z*Align pode trazer, dentro de um tempo aceitável, informações muito mais precisas acerca do grau de correlação entre dois organismos.

Capítulo 7

Conclusão e Trabalhos futuros

7.1 Conclusão

Esta dissertação apresentou uma estratégia paralela para a obtenção de alinhamentos ótimos através de comparações exatas entre seqüências biológicas longas em um espaço limitado de memória.

A estratégia obteve êxito na comparação das seqüências testadas, cujos comprimentos variam de 1 Kbp até 3 Mbp, tendo sido produzidos os alinhamentos ótimos para estas seqüências. O comprimento das seqüências analisadas não se apresentou como fator impeditivo para que a estratégia apresentada produzisse os alinhamentos desejados.

A estratégia apresentada foi capaz de produzir alinhamentos entre seqüências de aproximadamente 1 Mbp em 1 hora e 43 minutos utilizando 16 processadores AMD AthlonMP 1900+. A obtenção de alinhamentos para as seqüências de 3 Mbp levou 13 horas e 31 minutos sobre a mesma plataforma.

Foram medidos os *speedups* relativos para execuções com 1, 2, 4, 8, e 16 processadores para as seqüências de até 500 Kbp. Os resultados obtidos foram muito satisfatórios, ocorrendo *speedups* relativos superlineares em alguns casos. Para o caso do alinhamento de 500 Kbp, foi observado um speedup de 16,5 na execução com 16 processadores.

As comparações com o BLAST mostraram que a estratégia comporta-se conforme o desejado, produzindo alinhamentos ótimos de comprimento até 43 vezes superior aos produzidos pelo BLAST. O custo de tempo da estratégia apresentada é recompensado pela qualidade dos alinhamentos encontrados.

7.2 Trabalhos futuros

A estratégia mostrada nesta dissertação deixa ainda em aberto várias possibilidades de melhoria. A possibilidade que pode ser considerada mais importante é a adaptação da estratégia para que esta forneça não apenas os alinhamentos de melhor *score*, mas forneça também alinhamentos que tenham um *score* próximo do ótimo.

Testes ainda devem ser conduzidos com a estratégia para avaliar qual o comportamento desta para seqüências com mais de 3 Mbp. Tais análises irão por à

prova a eficiência da estratégia e suscitar ainda mais melhorias.

Apesar do custo de tempo envolvido, medidas de *speedup* devem ser feitas para seqüências com mais de 500 Kbp, bem como para um número maior de processadores, para verificar exatamente a partir de que ponto a solução passa a apresentar problemas na sua escalabilidade.

Estudos mais aprofundados devem ser desenvolvidos para definir a correlação entre os fatores h e b de divisão dos blocos, e outras variáveis associadas ao problema, como é o caso do tamanho das seqüências analisadas, o número de processadores utilizados, velocidade da rede de comunicação, dentre outros.

A fase de obtenção dos alinhamentos precisa ser melhorada no que diz respeito ao balanceamento de carga. Nos casos analisados, apenas o alinhamento de 1 Mbp teve o desempenho sensivelmente prejudicado devido à distribuição ineficiente de trabalho nesta fase. Além disso, como foram produzidos apenas os alinhamentos ótimos, a quantidade de alinhamentos presentes na quarta fase foi sempre mínima. Modificar a estratégia para produzir outros alinhamentos além dos ótimos pode aumentar muito o número de alinhamentos a serem processados nesta última fase.

A criação de um programa interativo de visualização dos restultados obtidos pode facilitar a análise dos resultados obtidos. Tal programa disporia de uma interface gráfica onde seria exibida uma representação gráfica dos alinhamentos e o usuário poderia examinar, em vários níveis de detalhe, regiões dos alinhamentos exibidos.

Referências

[1] N. Ahmed, Y. Pan, A. Vandenberg, and Y. Sun. Parallel algorithm for multiple genome alignment on the grid environment. In *International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, USA, 2005.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–10, Oct 5 1990.

[3] S. Aluru, N. Futamura, and K. Mehrotra. Parallel biological sequence comparison using prefix computations. *Journal of Parallel and Distributed Computing*, 63(3):264–272, 2003.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capability. *Proceedings of AFIPS Computer Conference*, 30:483–485, 1967.

[5] R. B. Batista, D. N. Silva, A. C. M. A. de Melo, and L. Weigang. Using a dsm application to locally align dna sequences. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID2004)*, Chicago, USA, 2004.

[6] A. Boukerche, A. C. M. A. Melo, M. Ayala-Rincón, and T. M. Santana. Parallel strategies for local biological sequence alignment in a cluster of workstations. In *International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, USA, 2005.

[7] A. Boukerche, A. C. M. A. Melo, M. E. T. Walter, R. C. F. Melo, M. N. P. Santana, and R. B. Batista. A performance evaluation of a local dna sequence alignment algorithm on a cluster of workstations. In *International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, USA, 2004.

[8] J. H. Carlson, S. F. Porcella, G. McClarty, and H. D. Caldwell. Comparative genomic analysis of chlamydia trachomatis oculotropic and genitotropic strains. In *Infection and Immunity*, volume 73, pages 6407–6418, Oct 2005.

[9] K. Charter, J. Schaeffer, and D. Szafron. Sequence alignment using fastlsa. In *International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences*, pages 239–245, June 2000.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[11] O. de Jesus, P. R. Smith, L. C. Spender, C. E. Karstegl, H. H. Niller, D. Huang, and P. J. Farrell. Updated epstein barr virus (ebv) dna sequence and analysis of a promoter for the bart (cst, barf0) rnas of ebv. In *Journal of General Virology*, volume 84, pages 1443–1450, 2003.

[12] J. B. del Cuvillo, W. S. Martins, G. R. Gao, W. Cui, and S. Kim. Atgc: Another tool for genome comparison. In *International Conference on Computational Molecular Biology(RECOMB'01)*, Montréal, Canada, April 2001.

[13] G. Dominguez, T. R. Dambaugh, F. R. Stamey, S. Dewhurst, N. Inoue, and P. E. Pellett. Human herpesvirus 6b genome sequence: Coding content and comparison with human herpesvirus 6a. In *Journal of Virology*, volume 73, pages 8040–8052, Oct 1999.

[14] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons. Fastlsa: A fast, linear-space, parallel and sequential algorithm for sequence alignment. In *International Conference on Parallel Processing(ICPP'03)*, pages 48–56, Kaohsiung, Taiwan, 2003.

[15] J. W. Fickett. Fast optimal alignments. *Nucleic Acids Research*, 12(1):175–179, 1984.

[16] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 30:948–960, september 1972.

[17] A. Fomenkov, J. P. Xiao, and S. Y. Xu. Nucleotide sequence of a small plasmid isolated from acetobacter pasteurianus. In *Gene*, volume 158, pages 143–144, 1995.

[18] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley Publishing Company, 1st edition, February 1994.

[19] C. Freiberg, R. Fellay, A. Bairoch, W. J. Broughton, A. Rosenthal, and X. Perret. Molecular basis of symbiosis between rhizobium and legumes. In *Nature*, volume 387, pages 394–401, May 1997.

[20] P. Gardner-Stephen and G. Knowles. Dash: Localising dynamic programming for order of magnitude faster, accurate sequence alignment. In *IEEE Computational Systems Bioinformatics Conference*, pages 732–735, Palo Alto, USA, 2004.

[21] B. Goodner and G. Hinkle. Genome sequence of the plant pathogen and biotechnology agent agrobacterium tumefaciens c58. In *Science*, volume 294, pages 2323–2328, Dec 2001.

[22] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[23] L. Grate and M. Diekhans. Sequence analysis with the kestrel simd parallel processor. In *Pacific Symposium on Biocomputing (PSB'01)*, volume 6, pages 263–274, Hawaii, USA, January 2001.

[24] D. S. Hirschberg. A linear space algorithm for computing longest common subsequences. *Communications of the Association of Computer Machinery*, 18(6):341–343, June 1975.

[25] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM system based on a new cache coherence protocol. In *High-Performance Computing and Networking Europe (HPCN'99)*, pages 463–472, 1999.

[26] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the earth-manna multithreaded system. In *International Journal of Parallel Programming*, volume 24, pages 319–348, New York, NY, USA, 1996. Plenum Press.

[27] L. Hunter. *Artificial Intelligence and Molecular Biology*. MIT Press, 1st edition, 1993.

[28] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, 1996.

[29] J. Kalinowski and B. Bathe et al. The complete corynebacterium glutamicum atcc 13032 genome sequence and its impact on the production of l-aspartate-derived amino acids and vitamins. In *Journal of Biotechnology*, volume 104, pages 5–25, 2003.

[30] A. Krishnan. Gridblast: A globus based high-throughput implementation of blast in a grid computing framework. *Pacific-Rim Symposium on Biocomputing*, 2003.

[31] A. M. Lesk. *Introduction to Bioinformatics*. Oxford University Press, 1st edition, 2002.

[32] W.G. Liu and B. Schmidt. Parallel design pattern for computational biology and scientific computing applications. In *Cluster 2003*, Hongkong, 2003.

[33] W. Martins, J. del Cuvillo, F. Useche, K. Theobald, and G. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison, 2001.

[34] W. S. Martins, J. B. del Cuvillo, W. Cui, and G. R. Gao. Whole genome alignment using a multithreaded parallel implementation. In *Symposium on Computer Architecture and High Performance Computing, Pirenopolis, Brazil*, pages 1–8, 2001.

[35] R. C. F. Melo, M. E. T. Walter, A. C. M. A. Melo, and R. B. Batista. Parallel dna sequence alignment using a dsm system in a cluster of workstations. In *International Conference on Computational Science(ICCS'03)*, San Petesburg, Russia, 2003.

[36] R. C. F. Melo, M. E. T. Walter, A. C. M. A. Melo, R. B. Batista, M. Nardelli, T. Martins, and T. Fonseca. Comparing two long biological sequences using a dsm system. *Lecture Notes in Computer Science*, 2790:517–524, August 2003.

[37] C. Mueller, M. Dalkilic, and A. Lumsdaine. High-performance direct pairwise comparison of large genomic sequences. In *Proceedings of the Fourth IEEE International Workshop on High Performance Computational Biology(HiCOMB'05)*, Denver, USA, April 2005.

[38] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988.

[39] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[40] J. R. Neilson and G. C. John et al. Subtypes of human immunodeficiency virus type 1 and disease stage among women in nairobi, kenya. In *Journal of Virology*, volume 73, pages 4393–4403, 1990.

[41] Y. Nishio, Y. Nakamura, Y. Kawarabayasi, Y. Usuda, and E. Kimura. Comparative complete genome sequence analysis of the amino acid replacements responsible for the thermostability of corynebacterium efficiens. In *Genome Research*, volume 13, pages 1572–1579, 2003.

[42] B. Paquin and B. F. Lang. The mitochondrial dna of allomyces macrogynus: the complete genomic sequence from an ancestral fungus. In *Journal of Molecular Biology*, volume 255, pages 688–701, Feb 1996.

[43] W. R. Pearson and L. D. Lipman. Improved tools for biological sequence comparison. *Proceedings Of The National Academy of Science USA*, 85:2444–2448, April 1988.

[44] S. Pellicer, N. Ahmed, Y. Pan, and Y. Zheng. Gene sequence alignment on a public computing platform. In *International Conference on Parallel Processing Workshops(ICPP'05)*, pages 95–102, Oslo, Norway, 2005.

[45] G. Pfister. *In Search of Clusters*. Prentice Hall PTR, 2nd edition, 1998.

[46] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. In *International Conference on Parallel Processing(ICPP'03)*, pages 39–47, 2003.

[47] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Pub. Co., 1997.

[48] J. C. Smith and A. C. Parker. A gene product related to tral is required for the mobilization of bacteroides mobilizable transposons and plasmids. In *Molecular Microbiology*, volume 20, pages 741–750, 1996.

[49] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[50] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1st edition, August 25 1994.

[51] T. D. Read TD and R. C. Brunham et al. Genome sequences of chlamydia trachomatis mopn and chlamydia pneumoniae ar39. In *Nucleic Acids Research*, volume 28, pages 1397–1406, 2000.

[52] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montreal, Québec, May 1999.

[53] M. Turmel, C. Otis, and C. Lemieux. The chloroplast and mitochondrial genome sequences of the charophyte chaetosphaeridium globosum: insights into the timing of the events that restructured organelle dnas within the green algal lineage that led to land plants. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 99, pages 11275–11280, Aug 2002.

[54] A. Vrenios. *Linux Cluster Architecture*. Sams Publishing, 2002.

[55] M. S. Waterman. *Mathematical Methods for DNA Sequences*. CRC Press Inc, Boca Ratón, Florida, 1989.

[56] M. S. Waterman and T. F. Smith. Some biological sequence metrics. *Advances in Mathematics*, 20:367–387, 1976.

[57] D. York-Higgins, C. Cheng-Mayer, D. Bauer, J. A. Levy, and D. Dina. Human immunodeficiency virus type 1 cellular host range, replication, and cytopathicity are linked to the envelope region of the viral genome. In *Journal of Virology*, volume 64, pages 4016–4020, 1990.

[58] F. Zhang, X. Qiao, and Z. Liu. A parallel smith-waterman algorithm based on divide and conquer. In *Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'02)*, 2003.